

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Java 并发编程

核心方法与框架

Java Concurrent Programming
Core Method and Frameworks

高洪岩 著



机械工业出版社
China Machine Press

技术畅销书《Java多线程编程核心技术》

作者撰写，全程案例式讲解，全面介绍Java并发包相关的技术。以浅白的措辞，结合大量实例模拟实际应用场景，全面解析Java并发包中的核心类、API与并发框架的使用。

全书共10章。第1章讲解了线程间的同步性，以及线程间的传输数据控制，即Semaphore和Exchanger类的使用。第2章介绍了在同步处理上更加灵活的工具类CountDownLatch和CyclicBarrier，详细到每个类的API的具体使用与应用场景。第3章是第2章的升级，由于CountDownLatch和CyclicBarrier类都有相应的弊端，所以在JDK 1.7中新增加了Phaser类来解决这些缺点，该类是熟练掌握JDK并发包的必要知识点。第4章是读者应重点掌握的Executor接口与ThreadPoolExecutor线程池，能有效地提高程序运行效率，更好地统筹线程执行的相关任务。第5章讲解Future和Callable的使用，解决线程需要返回值的情况。第6章介绍Java并发包中的CompletionService的使用，因为可以以异步的方式获得任务执行的结果，所以该接口可以增强程序运行效率。第7章介绍接口ExecutorService，该接口提供了若干工具方法来方便执行并发业务。第8章主要介绍ScheduledExecutorService的使用，以掌握如何将计划任务与线程池结合使用。第9章主要介绍Fork-Join分治编程，以提升多核CPU的优势，加快程序运行效率。第10章主要介绍并发集合框架，利用好并发框架，事半功倍。



Java 并发编程

核心方法与框架

Java Concurrent Programming
Core Method and Frameworks

高洪岩 著



本书特色

本书仅数页，少而精，是Java并发编程的入门书，也是Java并发编程的进阶书。本书从最基础的内容开始，逐步深入到高级内容，为读者提供了一条全面的思路。

本书特色

- 本书仅数页，少而精，是Java并发编程的入门书，也是Java并发编程的进阶书。
- 本书从最基础的内容开始，逐步深入到高级内容，为读者提供了一条全面的思路。
- 本书采用最简洁的语言，用最易懂的方式，让读者快速掌握Java并发编程的核心知识。
- 本书采用最实用的案例，让读者在实战中掌握Java并发编程的核心知识。
- 本书采用最清晰的结构，让读者在阅读时能够快速找到所需内容。
- 本书采用最丰富的插图，让读者在阅读时能够快速理解核心知识。



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Java 并发编程: 核心方法与框架 / 高洪岩著. —北京: 机械工业出版社, 2016.5
(Java 核心技术系列)

ISBN 978-7-111-53521-8

I. J… II. 高… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2016) 第 078593 号

Java 并发编程: 核心方法与框架

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 高婧雅

责任校对: 殷虹

印刷: 北京市荣盛彩色印刷有限公司

版次: 2016 年 5 月第 1 版第 1 次印刷

开本: 186mm × 240mm 1/16

印张: 23

书号: ISBN 978-7-111-53521-8

定价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Preface 前言

为什么要写这本书

早在几年前笔者就曾想过整理一份与 Java 并发包有关的稿件。因为市面上所有的 Java 书籍都是以 1 章或 2 章的篇幅介绍 Java 并发包技术，这就导致对 Java 并发包的讲解并不是非常详尽，包含的知识量远远不够，并没有完整覆盖 Java 并发包技术的知识点。但可惜，苦于当时的时间及精力有限，一直没有如愿。

也许是注定的安排，笔者现所在单位是集技术与教育为一体的软件类企业，学员在学习完 JavaSE/JavaEE 之后想探索更深入的技术，比如大数据、分布式、高并发类的专题，就会立即遇到与 Java 并发包中 API 相关的问题。为了带领学员在技术层面上有更高的追求，所以我将 Java 并发包的技术点以教案的方式进行整理，在课堂上与同学们一起进行学习、交流，同学们反响非常强烈。至此，若干年前的心愿终于了却，同学们也很期待这样一本书能出版发行，那样他们就有真正的纸质参考资料了。若这份资料也被其他爱好 Java 并发的朋友们看到，并通过它学到相关知识，那就是我最大的荣幸了。

本书将给读者一个完整的视角，秉承“大道至简”的主导思想，只介绍 Java 并发包开发中最值得关注的内容，希望能抛砖引玉，以个人的一些想法和见解，为读者拓展出更深入、全面的思路。

本书特色

本书尽量减少“啰嗦”式的文字语言，全部用 Demo 式案例来讲解技术点的实现，使读者看到代码及运行结果后就可以知道此项目要解决的是什么问题。类似于网络中 Blog 的风格，可让读者用最短的时间学会此知识点，明白此知识点如何应用，以及在使用时要避免什么。这就像“瑞士军刀”，虽短小，却锋利。本书的目的就是帮读者快速学习并解决问题。

读者对象

- ☐ Java 初级、中级程序员
- ☐ Java 多线程开发者
- ☐ Java 并发开发者
- ☐ 系统架构师
- ☐ 大数据开发者
- ☐ 其他对多线程技术感兴趣的人员

如何阅读本书

在整理本书时,笔者本着实用、易懂的学习原则整理了 10 个章节来介绍 Java 并发包相关的技术。

第 1 章讲解了 Semaphore 和 Exchanger 类的使用,学完本章后,能更好地控制线程间的同步性,以及线程间如何更好、更方便地传输数据。

第 2 章是第 1 章的延伸,主要讲解了 CountdownLatch、CyclicBarrier 类的使用及在 Java 并发包中对并发访问的控制。本章主要包括 Semaphore、CountDownLatch 和 CyclicBarrier 的使用,它们在使用上非常灵活,所以对于 API 的介绍比较详细,为读者学习控制同步打好坚实的基础。

第 3 章是第 2 章的升级,由于 CountdownLatch 和 CyclicBarrier 类都有相应的弊端,所以在 JDK1.7 中新增加了 Phaser 类来解决这些缺点。

第 4 章中讲解了 Executor 接口与 ThreadPoolExecutor 线程池的使用,可以说本章中的知识也是 Java 并发包中主要的应用技术点,线程池技术也在众多的高并发业务环境中使用。掌握线程池能更有效地提高程序运行效率,更好地统筹线程执行的相关任务。

第 5 章中讲解 Future 和 Callable 的使用,接口 Runnable 并不支持返回值,但在有些情况下真的需要返回值,所以 Future 就是用来解决这样的问题的。

第 6 章介绍 Java 并发包中的 CompletionService 的使用,该接口可以增强程序运行效率,因为可以以异步的方式获得任务执行的结果。

第 7 章主要介绍的是 ExecutorService 接口,该接口提供了若干方法来方便地执行业务,是比较常见的工具接口对象。

第 8 章主要介绍计划任务 ScheduledExecutorService 的使用,学完本章可以掌握如何将计划任务与线程池结合使用。

第 9 章主要介绍 Fork-Join 分治编程。分治编程在多核计算机中应用很广,它可以大大

的任务拆分成小的任务再执行，最后再把执行的结果聚合到一起，完全利用多核 CPU 的优势，加快程序运行效率。

第 10 章主要介绍并发集合框架。Java 中的集合在开发项目时占有举足轻重的地位，在 Java 并发包中也提供了在高并发环境中使用的 Java 集合工具类，读者需要着重掌握 Queue 接口的使用。

勘误和支持

由于笔者的水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。笔者邮箱是 279377921@qq.com，期待能够得到你们的真挚反馈，在技术之路上互勉共进。

本书的源代码可以在华章网站 (www.hzbook.com) 下载。

致谢

感谢所在单位领导的支持与厚爱，使我在技术道路上更有信心。

感谢机械工业出版社华章公司的编辑们始终支持我的写作，是你们的鼓励和帮助引导我顺利完成全部书稿。

高洪岩

目 录 Contents

前言

第 1 章 Semaphore 和 Exchanger 的使用	1
1.1 Semaphore 的使用	2
1.1.1 类 Semaphore 的同步性	2
1.1.2 类 Semaphore 构造方法 permits 参数作用	4
1.1.3 方法 acquire(int permits) 参数作用及动态添加 permits 许可数量	5
1.1.4 方法 acquireUninterruptibly() 的使用	8
1.1.5 方法 availablePermits() 和 drainPermits()	10
1.1.6 方法 getQueueLength() 和 hasQueuedThreads()	12
1.1.7 公平与非公平信号量的测试	13
1.1.8 方法 tryAcquire() 的使用	15
1.1.9 方法 tryAcquire(int permits) 的使用	17
1.1.10 方法 tryAcquire(long timeout, TimeUnit unit) 的使用	17
1.1.11 方法 tryAcquire(int permits, long timeout, TimeUnit unit) 的使用	19
1.1.12 多进路 - 多处理 - 多出路实验	20
1.1.13 多进路 - 单处理 - 多出路实验	21
1.1.14 使用 Semaphore 创建字符串池	23
1.1.15 使用 Semaphore 实现多生产者 / 多消费者模式	25
1.2 Exchanger 的使用	31
1.2.1 方法 exchange() 阻塞的特性	31
1.2.2 方法 exchange() 传递数据	32

1.2.3 方法 <code>exchange(V x, long timeout, TimeUnit unit)</code> 与超时	34
1.3 本章总结	35
第2章 CountdownLatch 和 CyclicBarrier 的使用	36
2.1 CountdownLatch 的使用	36
2.1.1 初步使用	37
2.1.2 裁判在等全部的运动员到来	38
2.1.3 各就各位准备比赛	39
2.1.4 完整的比赛流程	41
2.1.5 方法 <code>await(long timeout, TimeUnit unit)</code>	44
2.1.6 方法 <code>getCount()</code> 的使用	46
2.2 CyclicBarrier 的使用	46
2.2.1 初步使用	48
2.2.2 验证屏障重置性及 <code>getNumberWaiting()</code> 方法的使用	51
2.2.3 用 CyclicBarrier 类实现阶段跑步比赛	52
2.2.4 方法 <code>isBroken()</code> 的使用	55
2.2.5 方法 <code>await(long timeout, TimeUnit unit)</code> 超时出现异常的测试	57
2.2.6 方法 <code>getNumberWaiting()</code> 和 <code>getParties()</code>	60
2.2.7 方法 <code>reset()</code>	62
2.3 本章总结	64
第3章 Phaser 的使用	65
3.1 Phaser 的使用	66
3.2 类 Phaser 的 <code>arriveAndAwaitAdvance()</code> 方法测试 1	66
3.3 类 Phaser 的 <code>arriveAndAwaitAdvance()</code> 方法测试 2	68
3.4 类 Phaser 的 <code>arriveAndDeregister()</code> 方法测试	69
3.5 类 Phaser 的 <code>getPhase()</code> 和 <code>onAdvance()</code> 方法测试	70
3.6 类 Phaser 的 <code>getRegisteredParties()</code> 方法和 <code>register()</code> 测试	74
3.7 类 Phaser 的 <code>bulkRegister()</code> 方法测试	75
3.8 类 Phaser 的 <code>getArrivedParties()</code> 和 <code>getUnarrivedParties()</code> 方法测试	75
3.9 类 Phaser 的 <code>arrive()</code> 方法测试 1	77

3.10 类 Phaser 的 arrive () 方法测试 2	78
3.11 类 Phaser 的 awaitAdvance(int phase) 方法测试	81
3.12 类 Phaser 的 awaitAdvanceInterruptibly(int) 方法测试 1	83
3.13 类 Phaser 的 awaitAdvanceInterruptibly(int) 方法测试 2	84
3.14 类 Phaser 的 awaitAdvanceInterruptibly(int) 方法测试 3	86
3.15 类 Phaser 的 awaitAdvanceInterruptibly(int,long,TimeUnit) 方法测试 4	87
3.16 类 Phaser 的 forceTermination() 和 isTerminated() 方法测试	89
3.17 控制 Phaser 类的运行时机	92
3.18 本章总结	93

第 4 章 Executor 与 ThreadPoolExecutor 的使用

4.1 Executor 接口介绍	94
4.2 使用 Executors 工厂类创建线程池	97
4.2.1 使用 newCachedThreadPool() 方法创建无界线程池	98
4.2.2 验证 newCachedThreadPool() 创建为 Thread 池	100
4.2.3 使用 newCachedThreadPool (ThreadFactory) 定制线程工厂	102
4.2.4 使用 newFixedThreadPool(int) 方法创建有界线程池	103
4.2.5 使用 newFixedThreadPool(int, ThreadFactory) 定制线程工厂	105
4.2.6 使用 newSingleThreadExecutor() 方法创建单一线程池	106
4.2.7 使用 newSingleThreadExecutor(ThreadFactory) 定制线程工厂	107
4.3 ThreadPoolExecutor 的使用	107
4.3.1 构造方法的测试	107
4.3.2 方法 shutdown() 和 shutdownNow() 与返回值	119
4.3.3 方法 isShutdown()	129
4.3.4 方法 isTerminating () 和 isTerminated ()	129
4.3.5 方法 awaitTermination(long timeout,TimeUnit unit)	131
4.3.6 工厂 ThreadFactory+execute()+UncaughtExceptionHandler 处理异常	134
4.3.7 方法 set/getRejectedExecutionHandler()	138
4.3.8 方法 allowsCoreThreadTimeOut()/(boolean)	140
4.3.9 方法 prestartCoreThread() 和 prestartAllCoreThreads()	142
4.3.10 方法 getCompletedTaskCount()	144

4.3.11 常见 3 种队列结合 max 值的因果效果	145
4.3.12 线程池 ThreadPoolExecutor 的拒绝策略	151
4.3.13 方法 afterExecute() 和 beforeExecute()	157
4.3.14 方法 remove(Runnable) 的使用	159
4.3.15 多个 get 方法的测试	162
4.3.16 线程池 ThreadPoolExecutor 与 Runnable 执行为乱序特性	166
4.4 本章总结	167
第 5 章 Future 和 Callable 的使用	168
5.1 Future 和 Callable 的介绍	168
5.2 方法 get() 结合 ExecutorService 中的 submit(Callable<T>) 的使用	168
5.3 方法 get() 结合 ExecutorService 中的 submit(Runnable) 和 isDone() 的使用	170
5.4 使用 ExecutorService 接口中的方法 submit(Runnable, T result)	170
5.5 方法 cancel(boolean mayInterruptIfRunning) 和 isCancelled() 的使用	173
5.6 方法 get(long timeout, TimeUnit unit) 的使用	178
5.7 异常的处理	179
5.8 自定义拒绝策略 RejectedExecutionHandler 接口的使用	181
5.9 方法 execute() 与 submit() 的区别	182
5.10 验证 Future 的缺点	186
5.11 本章总结	188
第 6 章 CompletionService 的使用	189
6.1 CompletionService 介绍	189
6.2 使用 CompletionService 解决 Future 的缺点	190
6.3 使用 take() 方法	193
6.4 使用 poll() 方法	194
6.5 使用 poll(long timeout, TimeUnit unit) 方法	195
6.6 类 CompletionService 与异常	199
6.7 方法 Future<V> submit(Runnable task, V result) 的测试	205
6.8 本章总结	207

第7章 接口 ExecutorService 的方法使用	208
7.1 在 ThreadPoolExecutor 中使用 ExecutorService 中的方法	208
7.2 方法 invokeAny(Collection tasks) 的使用与 interrupt	209
7.3 方法 invokeAny() 与执行慢的任务异常	212
7.4 方法 invokeAny() 与执行快的任务异常	216
7.5 方法 invokeAny() 与全部异常	220
7.6 方法 invokeAny(CollectionTasks, timeout, timeUnit) 超时的测试	222
7.7 方法 invokeAll(Collection tasks) 全正确	226
7.8 方法 invokeAll(Collection tasks) 快的正确慢的异常	227
7.9 方法 invokeAll(Collection tasks) 快的异常慢的正确	230
7.10 方法 invokeAll(Collection tasks, long timeout, TimeUnit unit) 先慢后快	232
7.11 方法 invokeAll(Collection tasks, long timeout, TimeUnit unit) 先快后慢	234
7.12 方法 invokeAll(Collection tasks, long timeout, TimeUnit unit) 全慢	236
7.13 本章总结	238
第8章 计划任务 ScheduledExecutorService 的使用	239
8.1 ScheduledExecutorService 的使用	240
8.2 ScheduledThreadPoolExecutor 使用 Callable 延迟运行	241
8.3 ScheduledThreadPoolExecutor 使用 Runnable 延迟运行	244
8.4 延迟运行并取得返回值	245
8.5 使用 scheduleAtFixedRate() 方法实现周期性执行	246
8.6 使用 scheduleWithFixedDelay() 方法实现周期性执行	248
8.7 使用 getQueue() 与 remove() 方法	250
8.8 方法 setExecuteExistingDelayedTasksAfterShutdownPolicy() 的使用	253
8.9 方法 setContinueExistingPeriodicTasksAfterShutdownPolicy()	255
8.10 使用 cancel(boolean) 与 setRemoveOnCancelPolicy() 方法	257
8.11 本章总结	261
第9章 Fork-Join 分治编程	262
9.1 Fork-Join 分治编程与类结构	262
9.2 使用 RecursiveAction 让任务跑起来	264

9.3 使用 RecursiveAction 分解任务	265
9.4 使用 RecursiveTask 取得返回值与 join() 和 get() 方法的区别	266
9.5 使用 RecursiveTask 执行多个任务并打印返回值	270
9.6 使用 RecursiveTask 实现字符串累加	272
9.7 使用 Fork-Join 实现求和：实验 1	273
9.8 使用 Fork-Join 实现求和：实验 2	275
9.9 类 ForkJoinPool 核心方法的实验	276
9.9.1 方法 public void execute(ForkJoinTask<?> task) 的使用	276
9.9.2 方法 public void execute(Runnable task) 的使用	278
9.9.3 方法 public void execute(ForkJoinTask<?> task) 如何处理返回值	278
9.9.4 方法 public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task) 的使用	279
9.9.5 方法 public ForkJoinTask<?> submit(Runnable task) 的使用	280
9.9.6 方法 public <T> ForkJoinTask<T> submit(Callable<T> task) 的使用	281
9.9.7 方法 public <T> ForkJoinTask<T> submit(Runnable task, T result) 的使用	282
9.9.8 方法 public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) 的使用	285
9.9.9 方法 public void shutdown() 的使用	286
9.9.10 方法 public List<Runnable> shutdownNow() 的使用	289
9.9.11 方法 isTerminating() 和 isTerminated() 的使用	292
9.9.12 方法 public boolean isShutdown() 的使用	295
9.9.13 方法 public boolean awaitTermination(long timeout, TimeUnit unit) 的使用	297
9.9.14 方法 public <T> T invoke(ForkJoinTask<T> task) 的使用	299
9.9.15 监视 pool 池的状态	301
9.10 类 ForkJoinTask 对异常的处理	308
9.11 本章总结	309

第 10 章 并发集合框架

10.1 集合框架结构简要	310
10.1.1 接口 Iterable	310
10.1.2 接口 Collection	311
10.1.3 接口 List	311
10.1.4 接口 Set	312

10.1.5	接口 Queue	312
10.1.6	接口 Deque	312
10.2	非阻塞队列	313
10.2.1	类 ConcurrentHashMap 的使用	313
10.2.2	类 ConcurrentSkipListMap 的使用	322
10.2.3	类 ConcurrentSkipListSet 的使用	325
10.2.4	类 ConcurrentLinkedQueue 的使用	328
10.2.5	类 ConcurrentLinkedDeque 的使用	330
10.2.6	类 CopyOnWriteArrayList 的使用	332
10.2.7	类 CopyOnWriteArraySet 的使用	335
10.3	阻塞队列	337
10.3.1	类 ArrayBlockingQueue 的使用	337
10.3.2	类 PriorityBlockingQueue 的使用	338
10.3.3	类 LinkedBlockingQueue 的使用	340
10.3.4	类 LinkedBlockingDeque 的使用	341
10.3.5	类 SynchronousQueue 的使用	341
10.3.6	类 DelayQueue 的使用	344
10.3.7	类 LinkedTransferQueue 的使用	345
10.4	本章总结	354

Semaphore 和 Exchanger 的使用

本书将介绍并发包中常见的并发类的主要 API 方法，掌握这些 API 方法所提供的功能是掌握并发包技术的主要手段，每一个类所提供的功能都是独有的，控制线程的行为也是不同的，这些都要依赖于类中的方法才可以实现。并发工具类中的方法其实并不算少，但它们之间却有着非常相似的功能，所以在学习上可以增加效率，理解起来并不是非常复杂。

作为本书的第 1 章，我将和大家一起交流一下类 Semaphore 和 Exchanger 的使用及其有关 API，类 Semaphore 所提供的功能完全就是 synchronized 关键字的升级版，但它提供的功能更加的强大与方便，主要的作用就是控制线程并发的数量，而这一点，单纯地使用 synchronized 是做不到的。

在本章将介绍 Semaphore 类中的常用 API，方法列表如图 1-1 所示。

类 Exchanger 的主要作用可以使 2 个线程之间互相方便地进行通信，它的常用 API 如图 1-2 所示。

```

● acquire() : void - Semaphore
● acquire(int permits) : void - Semaphore
● acquireUninterruptibly() : void - Semaphore
● acquireUninterruptibly(int permits) : void - Semaphore
● availablePermits() : int - Semaphore
● drainPermits() : int - Semaphore
● equals(Object obj) : boolean - Object
● getClass() : Class<?> - Object
● getQueueLength() : int - Semaphore
● hashCode() : int - Object
● hasQueuedThreads() : boolean - Semaphore
● isFair() : boolean - Semaphore
● notify() : void - Object
● notifyAll() : void - Object
● release() : void - Semaphore
● release(int permits) : void - Semaphore
● toString() : String - Semaphore
● tryAcquire() : boolean - Semaphore
● tryAcquire(int permits) : boolean - Semaphore
● tryAcquire(long timeout, TimeUnit unit) : boolean - Semaphore
● tryAcquire(int permits, long timeout, TimeUnit unit) : boolean - Semaphore
● wait() : void - Object
● wait(long timeout) : void - Object
● wait(long timeout, int nanos) : void - Object

```

图 1-1 类 Semaphore 中的 API

```

equals(Object obj) : boolean - Object
exchange(Object x) : Object - Exchanger
exchange(Object x, long timeout, TimeUnit unit) : Object - Exchanger
getClass() : Class<?> - Object
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
toString() : String - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object

```

图 1-2 类 Exchanger 中的 API

1.1 Semaphore 的使用

本章将对 Semaphore 类中的全部方法进行案例式的实验，这样可以全面地了解此类提供了哪些核心功能。

单词 Semaphore[ˈseməˈfoʊ(r)] 的中文含义是信号、信号系统。此类的主要作用就是限制线程并发的数量，如果不限制线程并发的数量，则 CPU 的资源很快就被耗尽，每个线程执行的任务是相当缓慢，因为 CPU 要把时间片分配给不同的线程对象，而且上下文切换也要耗时，最终造成系统运行效率大幅降低，所以限制并发线程的数量还是非常必要的。

在生活中也存在这种场景，比如一个生产键盘的生产商，发布了 10 个代理销售许可，所以最多只有 10 个代理商来获得其中的一个许可，这样就限制了代理商的数量，同理也限制了线程并发数的数量，这就是 Semaphore 类要达到的目的。

Semaphore 类发放许可的计算方式是“减法”操作。

1.1.1 类 Semaphore 的同步性

多线程中的同步概念其实就是排着队去执行一个任务，执行任务是一个一个执行，并不能并行执行，这样的优点是有助于程序逻辑的正确性，不会出现非线性安全问题，保证软件系统功能上的运行稳定性。

那么本节就使用一个初步的案例来看看 Semaphore 类是如何实现限制线程并发数的。

创建实验用的项目 SemaphoreTest1，类 Service.java 代码如下：

```

package service;

import java.util.concurrent.Semaphore;

public class Service {

    private Semaphore semaphore = new Semaphore(1);

    public void testMethod() {
        try {
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName())

```

```

        + " begin timer=" + System.currentTimeMillis();
        Thread.sleep(5000);
        System.out.println(Thread.currentThread().getName()
            + "    end timer=" + System.currentTimeMillis());
        semaphore.release();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

类 Semaphore 的构造函数参数 permits 是许可的意思, 代表同一时间内, 最多允许多少个线程同时执行 acquire() 和 release() 之间的代码。

无参方法 acquire() 的作用是使用 1 个许可, 是减法操作。

创建 3 个线程类如图 1-3 所示。

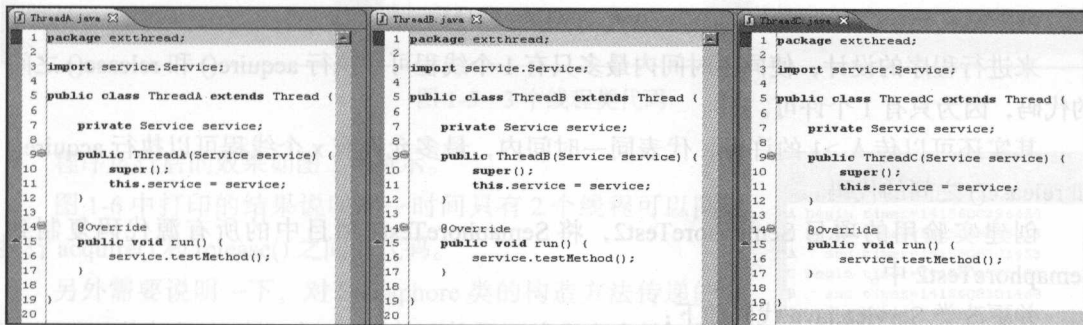


图 1-3 线程数量为 3

运行类 Run.java 代码如下:

```

package test;

import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
import extthread.ThreadC;

public class Run {

    public static void main(String[] args) {
        Service service = new Service();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        ThreadB b = new ThreadB(service);
        b.setName("B");
        ThreadC c = new ThreadC(service);
        c.setName("C");
        a.start();
        b.start();
    }
}

```



```

        c.start();
    }
}

```

程序运行后的效果如图 1-4 所示。

说明使用代码：

```
private Semaphore semaphore = new Semaphore(1);
```

来定义最多允许 1 个线程执行 `acquire()` 和 `release()` 之间的代码，所以打印的结果就是 3 个线程是同步的。

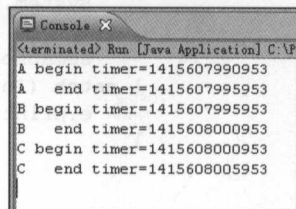


图 1-4 同步运行了

1.1.2 类 Semaphore 构造方法 permits 参数作用

参数 `permits` 的作用是设置许可的个数，前面已经使用过代码：

```
private Semaphore semaphore = new Semaphore(1);
```

来进行程序的设计，使同一时间内最多只有 1 个线程可以执行 `acquire()` 和 `release()` 之间的代码，因为只有 1 个许可。

其实还可以传入 >1 的许可，代表同一时间内，最多允许有 x 个线程可以执行 `acquire()` 和 `release()` 之间的代码。

创建实验用的项目 `SemaphoreTest2`，将 `SemaphoreTest1` 项目中的所有源代码复制到 `SemaphoreTest2` 中。

并更改类 `Service.java` 代码如下：

```

package service;

import java.util.concurrent.Semaphore;

public class Service {

    private Semaphore semaphore = new Semaphore(2);

    public void testMethod() {
        try {
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName()
                + " begin timer=" + System.currentTimeMillis());
            Thread.sleep(5000);
            System.out.println(Thread.currentThread().getName()
                + " end timer=" + System.currentTimeMillis());
            semaphore.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

使用代码 `new Semaphore(2)` 来实例化 `Semaphore` 类的含义是同一时间内最多允许 2 个线程执行 `acquire()` 和 `release()` 之间的代码。

创建 3 个线程类如图 1-5 所示。

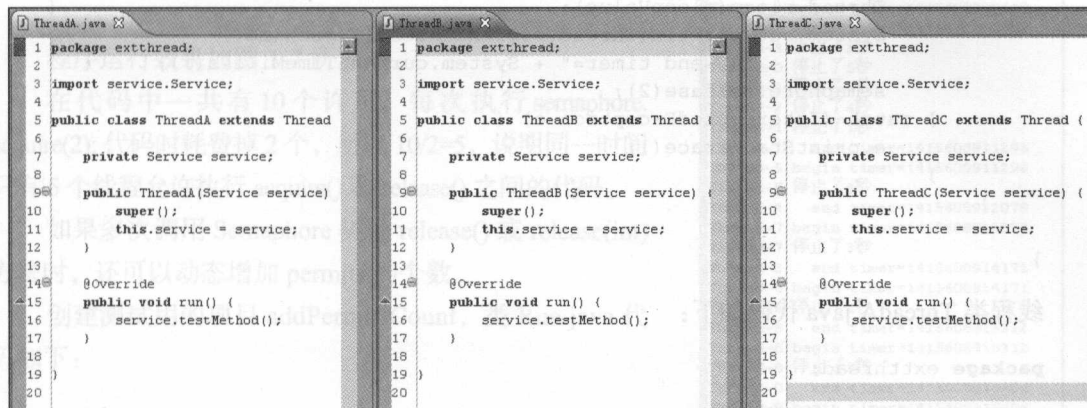


图 1-5 3 个线程类代码

程序运行后的效果如图 1-6 所示。

图 1-6 中打印的结果说明同一时间只有 2 个线程可以同时执行 `acquire()` 和 `release()` 之间的代码。

另外需要说明一下，对 `Semaphore` 类的构造方法传递的参数 `permits` 值如果大于 1 时，该类并不能保证线程安全性，因为还是有可能出现多个线程共同访问实例变量，导致出现脏数据的情况。

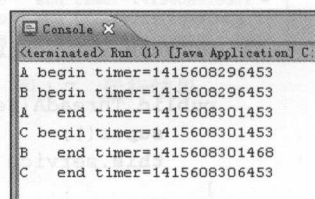


图 1-6 线程 AB 共同进入 C 被排斥

1.1.3 方法 `acquire(int permits)` 参数作用及动态添加 `permits` 许可数量

有参方法 `acquire(int permits)` 的功能是每调用 1 次此方法，就使用 `x` 个许可。

创建 Java 项目，名称为 `Semaphore_acquire(int permits)_release(int permits)`，创建类 `Service.java` 代码如下：

```

package service;

import java.util.concurrent.Semaphore;

public class Service {

    private Semaphore semaphore = new Semaphore(10);

    public void testMethod() {
        try {
            semaphore.acquire(2);
        }
    }
}
  
```

线程类 ThreadA.java 代码如下:

类 Run.java 代码如下:

```
package test;

import service.Service;
import extthread.ThreadA;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        Service service = new Service();

        ThreadA[] a = new ThreadA[10];
        for (int i = 0; i < 10; i++) {
            a[i] = new ThreadA(service);
        }
    }
}
```

```

        a[i].start();
    }

}

```

程序运行效果如图 1-7 所示。

在代码中一共有 10 个许可，每次执行 semaphore.acquire(2); 代码时耗费掉 2 个，所以 $10/2=5$ ，说明同一时间只有 5 个线程允许执行 acquire() 和 release() 之间的代码。

如果多次调用 Semaphore 类的 release() 或 release(int) 方法时，还可以动态增加 permits 的个数。

创建测试用的项目 addPermitsCount，类 Run.java 代码如下：

```


package test;

import java.util.concurrent.Semaphore;

public class Run {

    public static void main(String[] args) {
        try {
            Semaphore semaphore = new Semaphore(5);
            semaphore.acquire();
            semaphore.acquire();
            semaphore.acquire();
            semaphore.acquire();
            semaphore.acquire();
            System.out.println(semaphore.availablePermits());
            semaphore.release();
            semaphore.release();
            semaphore.release();
            semaphore.release();
            semaphore.release();
            System.out.println(semaphore.availablePermits());
            semaphore.release(4);
            System.out.println(semaphore.availablePermits());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```



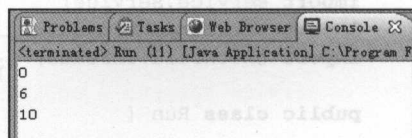
```

<terminated> Run (2) [Java Application] C:\Program
Thread-0 begin timer=1415608909609
Thread-4 begin timer=1415608909609
Thread-3 begin timer=1415608909609
Thread-2 begin timer=1415608909609
Thread-1 begin timer=1415608909609
Thread-4 停止了2秒
Thread-0 停止了5秒
Thread-3 停止了5秒
Thread-2 停止了4秒
Thread-1 停止了1秒
Thread-1 end timer=1415608911296
Thread-5 begin timer=1415608911296
Thread-5 停止了4秒
Thread-4 end timer=1415608912078
Thread-7 begin timer=1415608912078
Thread-7 停止了3秒
Thread-2 end timer=1415608914171
Thread-6 begin timer=1415608914171
Thread-6 停止了1秒
Thread-5 end timer=1415608915312
Thread-8 begin timer=1415608915312
Thread-8 停止了5秒
Thread-0 end timer=1415608915406
Thread-9 begin timer=1415608915406
Thread-9 停止了4秒
Thread-3 end timer=1415608915421
Thread-7 end timer=1415608915531
Thread-6 end timer=1415608915531
Thread-9 end timer=1415608920062
Thread-8 end timer=1415608921250

```

图 1-7 运行结果

程序运行后的效果如图 1-8 所示。



```

Problems Tasks Web Browser Console X
<terminated> Run (11) [Java Application] C:\Program F
0
6
10

```

图 1-8 成功动态添加许可

此实验说明构造参数 `new Semaphore(5);` 中的 5 并不是最终的许可数量，仅仅是初始的状态值。

1.1.4 方法 `acquireUninterruptibly()` 的使用

方法 `acquireUninterruptibly()` 的作用是使等待进入 `acquire()` 方法的线程，不允许被中断。先来看一个能中断的实验。

创建项目 `Semaphore_acquireUninterruptibly_1`，类 `Service.java` 代码如下：

```
package service;

import java.util.concurrent.Semaphore;

public class Service {
    private Semaphore semaphore = new Semaphore(1);

    public void testMethod() {
        try {
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName()
                + " begin timer=" + System.currentTimeMillis());
            for (int i = 0; i < Integer.MAX_VALUE / 50; i++) {
                String newString = new String();
                Math.random();
            }
            System.out.println(Thread.currentThread().getName()
                + " end timer=" + System.currentTimeMillis());
            semaphore.release();
        } catch (InterruptedException e) {
            System.out.println("线程 " + Thread.currentThread().getName()
                + " 进入了 catch");
            e.printStackTrace();
        }
    }
}
```

线程类代码如图 1-9 所示。

运行类 `Run.java` 代码如下：

```
package test;

import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;

public class Run {

    public static void main(String[] args) throws InterruptedException {
```



```

Service service = new Service();
ThreadA a = new ThreadA(service);
a.setName("A");
a.start();

ThreadB b = new ThreadB(service);
b.setName("B");
b.start();

Thread.sleep(1000);

b.interrupt();
System.out.println("main 中断了 a");
}
}

```

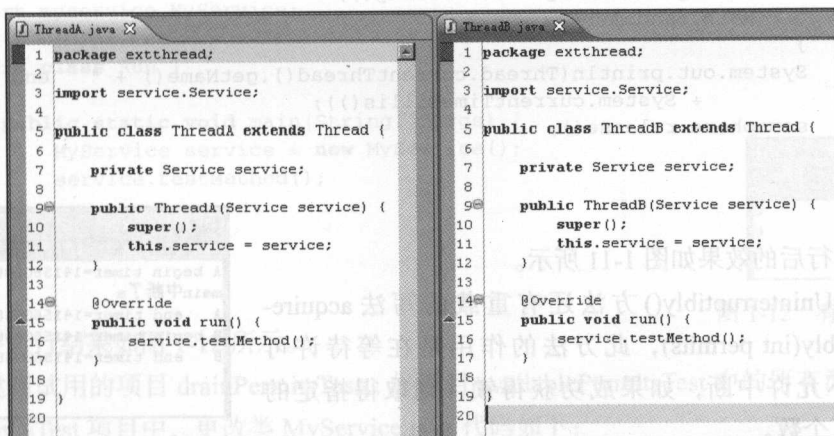


图 1-9 线程类代码

程序运行的效果如图 1-10 所示。

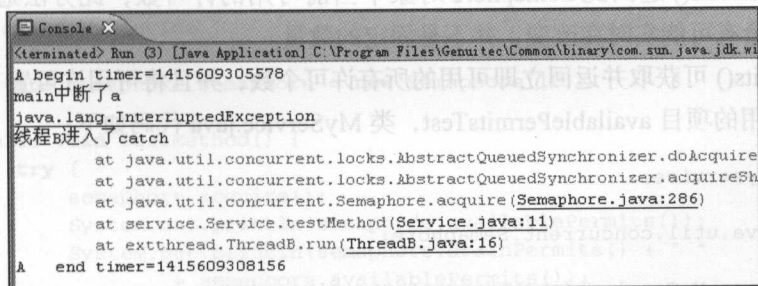


图 1-10 线程 B 被中断

线程 B 成功被中断。

那么不能被中断是什么效果呢？

创建项目 Semaphore_acquireUninterruptibly_2, 将 Semaphore_acquireUninterruptibly_1 项目中的所有源代码复制到 Semaphore_acquireUninterruptibly_2 中, 更改 Service.java 文件代码如下:

```
package service;

import java.util.concurrent.Semaphore;

public class Service {

    private Semaphore semaphore = new Semaphore(1);

    public void testMethod() {
        semaphore.acquireUninterruptibly();
        System.out.println(Thread.currentThread().getName() + " begin timer="
            + System.currentTimeMillis());
        for (int i = 0; i < Integer.MAX_VALUE / 50; i++) {
            String newString = new String();
            Math.random();
        }
        System.out.println(Thread.currentThread().getName() + " end timer="
            + System.currentTimeMillis());
        semaphore.release();
    }
}
```

程序运行后的效果如图 1-11 所示。

acquireUninterruptibly() 方法还有重载的写法 acquireUninterruptibly(int permits), 此方法的作用是在等待许可的情况下不允许中断, 如果成功获得锁, 则取得指定的 permits 许可个数。

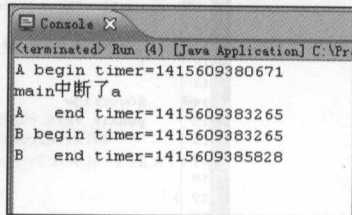


图 1-11 线程 B 没有被中断

1.1.5 方法 availablePermits() 和 drainPermits()

availablePermits() 返回此 Semaphore 对象中当前可用的许可数, 此方法通常用于调试, 因为许可的数量有可能实时在改变, 并不是固定的数量。

drainPermits() 可获取并返回立即可用的所有许可个数, 并且将可用许可置 0。

创建测试用的项目 availablePermitsTest, 类 MyService.java 代码如下:

```
package myservice;

import java.util.concurrent.Semaphore;

public class MyService {

    private Semaphore semaphore = new Semaphore(10);

    public void testMethod() {
        try {
```

```

        semaphore.acquire();
        System.out.println(semaphore.availablePermits());
        System.out.println(semaphore.availablePermits());
        System.out.println(semaphore.availablePermits());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        semaphore.release();
    }
}

```

类 Run.java 代码如下:

```

package test.run;

import myservice.MyService;

public class Run {

    public static void main(String[] args) {
        MyService service = new MyService();
        service.testMethod();
    }
}

```

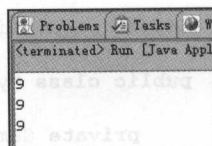


图 1-12 有效的许可数量

程序运行结果如图 1-12 所示。

创建测试用的项目 drainPermitsTest, 将项目 availablePermitsTest 中的所有源代码复制到 drainPermitsTest 项目中, 更改类 MyService.java 代码如下:

```

package myservice;

import java.util.concurrent.Semaphore;

public class MyService {

    private Semaphore semaphore = new Semaphore(10);

    public void testMethod() {
        try {
            semaphore.acquire();
            System.out.println(semaphore.availablePermits());
            System.out.println(semaphore.drainPermits() + " "
                + semaphore.availablePermits());
            System.out.println(semaphore.drainPermits() + " "
                + semaphore.availablePermits());
            System.out.println(semaphore.drainPermits() + " "
                + semaphore.availablePermits());
        } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    } finally {
        semaphore.release();
    }
}
}

```

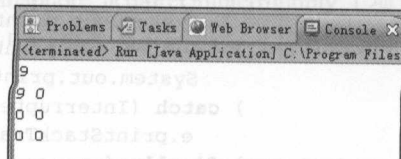


图 1-13 许可被置 0

程序运行后的效果如图 1-13 所示。

执行方法 `drainPermits()` 返回可用的许可个数，并将可用许可数清零。

1.1.6 方法 `getQueueLength()` 和 `hasQueuedThreads()`

方法 `getQueueLength()` 的作用是取得等待许可的线程个数。

方法 `hasQueuedThreads()` 的作用是判断有没有线程在等待这个许可。

这两个方法通常都是在判断当前有没有等待许可的线程信息时使用。

创建测试用的项目 `twoMethodTest`，类 `MyService.java` 代码如下：

```

package myservice;

import java.util.concurrent.Semaphore;

public class MyService {

    private Semaphore semaphore = new Semaphore(1);

    public void testMethod() {
        try {
            semaphore.acquire();
            Thread.sleep(1000);
            System.out.println(" 还有大约 " + semaphore.getQueueLength() + " 个线程在等待 ");
            System.out.println(" 是否有线程正在等待信号量呢？ " + semaphore.
                hasQueuedThreads());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release();
        }
    }
}

```

类 `MyThread.java` 代码如下：

```

package extthread;

import myservice.MyService;

public class MyThread extends Thread {

    private MyService myService;
}

```

```

public MyThread(MyService myService) {
    super();
    this.myService = myService;
}

@Override
public void run() {
    myService.testMethod();
}
}

```

类 Run.java 代码如下:

```

package test.run;

import myservice.MyService;
import extthread.MyThread;

public class Run {

    public static void main(String[] args) {
        MyService service = new MyService();

        MyThread firstThread = new MyThread(service);
        firstThread.start();

        MyThread[] threadArray = new MyThread[4];
        for (int i = 0; i < 4; i++) {
            threadArray[i] = new MyThread(service);
            threadArray[i].start();
        }
    }
}

```

程序运行后的效果如图 1-14 所示。

线程的个数呈递减的状态。

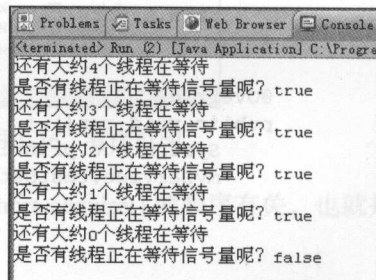


图 1-14 运行结果

1.1.7 公平与非公平信号量的测试

有些时候, 获得许可的顺序与线程启动的顺序有关, 这时信号量就要分为公平与非公平的。

所谓的公平信号量是获得锁的顺序与线程启动的顺序有关, 但不代表 100% 地获得信号量, 仅仅是在概率上能得到保证。而非公平信号量就是无关的了。

创建测试用的项目 semaphoreFairTest, 类 MyService.java 代码如下:

```

package myservice;

import java.util.concurrent.Semaphore;

public class MyService {

```



```

private boolean isFair = false;
private Semaphore semaphore = new Semaphore(1, isFair);

public void testMethod() {
    try {
        semaphore.acquire();
        System.out
            .println("ThreadName=" + Thread.currentThread().getName());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        semaphore.release();
    }
}

```

类 MyThread.java 代码如下：

```

package extthread;

import myservice.MyService;

public class MyThread extends Thread {

    private MyService myService;

    public MyThread(MyService myService) {
        super();
        this.myService = myService;
    }

    @Override
    public void run() {
        System.out.println("ThreadName=" + this.getName() + " 启动了！");
        myService.testMethod();
    }
}

```

类 Run.java 代码如下：

```

package test.run;

import myservice.MyService;
import extthread.MyThread;

public class Run {

    public static void main(String[] args) {
        MyService service = new MyService();

        MyThread firstThread = new MyThread(service);
        firstThread.start();
    }
}

```

```

MyThread[] threadArray = new MyThread[4];
for (int i = 0; i < 4; i++) {
    threadArray[i] = new MyThread(service);
    threadArray[i].start();
}
}
}

```

程序运行后的效果如图 1-15 所示。

非公平信号量运行的效果是线程启动的顺序与调用 `semaphore.acquire()` 的顺序无关，也就是线程先启动了并不代表先获得许可。

更改 `MyService.java` 类代码如下：

```
private boolean isFair = true;
```

程序运行结果如图 1-16 所示。

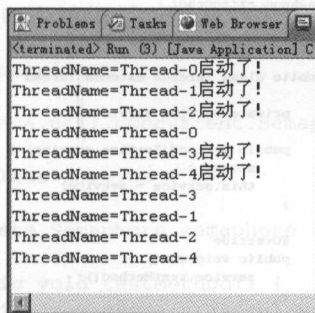


图 1-15 乱序打印

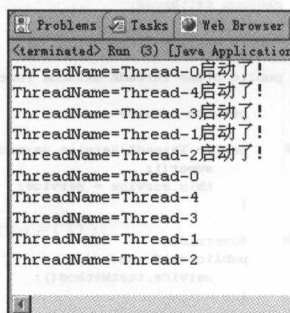


图 1-16 有序打印

公平信号量运行的效果是线程启动的顺序与调用 `semaphore.acquire()` 的顺序有关，也就是先启动的线程优先获得许可。

1.1.8 方法 `tryAcquire()` 的使用

无参方法 `tryAcquire()` 的作用是尝试地获得 1 个许可，如果获取不到则返回 `false`，此方法通常与 `if` 语句结合使用，其具有无阻塞的特点。无阻塞的特点可以使线程不至于在同步处一直持续等待的状态，如果 `if` 语句判断不成立则线程会继续走 `else` 语句，程序会继续向下运行。

创建 Java 项目 `Semaphore_tryAcquire_1`，类代码如下：

```

package service;

import java.util.concurrent.Semaphore;

public class Service {

```

```

private Semaphore semaphore = new Semaphore(1);

public void testMethod() {
    if (semaphore.tryAcquire()) {
        System.out.println("ThreadName=" + Thread.currentThread().getName()
            + " 首选进入! ");
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            String newString = new String();
            Math.random();
        }
        semaphore.release();
    } else {
        System.out.println("ThreadName=" + Thread.currentThread().getName()
            + " 未成功进入! ");
    }
}
}

```

两个线程类如图 1-17 所示。

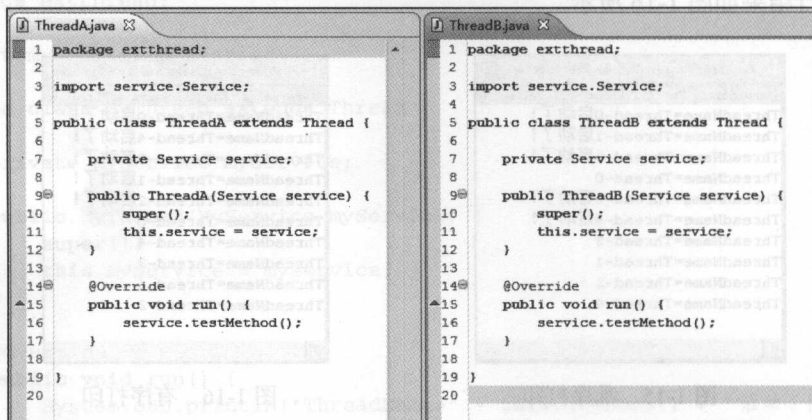


图 1-17 线程类代码

运行类 Run.java 代码如下：

```

package test.run;

import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;

public class Run {

    public static void main(String[] args) {
        Service service = new Service();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("B");
    }
}

```

```
b.start();
```

程序运行后的效果如图 1-18 所示。

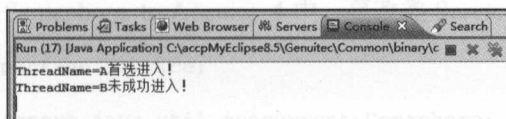


图 1-18 线程 B 未获得许可

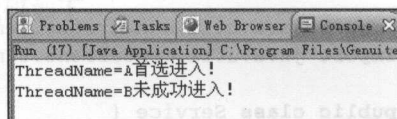


图 1-19 线程 B 未获得许可

1.1.9 方法 tryAcquire(int permits) 的使用

有参方法 tryAcquire(int permits) 的作用是尝试地获得 x 个许可，如果获取不到则返回 false。下面的项目就验证这个结论。

创建 Java 项目 Semaphore_tryAcquire_2，将项目 Semaphore_tryAcquire_1 中的源代码复制到 Semaphore_tryAcquire_2 中，更改类代码如下：

```
package service;

import java.util.concurrent.Semaphore;

public class Service {

    private Semaphore semaphore = new Semaphore(3);

    public void testMethod() {
        if (semaphore.tryAcquire(3)) {
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                + " 首选进入! ");
            for (int i = 0; i < Integer.MAX_VALUE; i++) {
                String newString = new String();
                Math.random();
            }
            // 方法 release 对应的 permits 值也要更改
            semaphore.release(3);
        } else {
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                + " 未成功进入! ");
        }
    }
}
```

程序运行后的效果如图 1-19 所示。

1.1.10 方法 tryAcquire(long timeout, TimeUnit unit) 的使用

有参方法 tryAcquire(long timeout, TimeUnit unit) 的作用是在指定的时间内尝试地获得

1 个许可，如果获取不到则返回 false。

创建 Java 项目 Semaphore_tryAcquire_3，将项目 Semaphore_tryAcquire_2 中的源代码复制到 Semaphore_tryAcquire_3 中，更改类代码如下：

```
package service;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

public class Service {

    private Semaphore semaphore = new Semaphore(1);

    public void testMethod() {
        try {
            if (semaphore.tryAcquire(3, TimeUnit.SECONDS)) {
                System.out.println("ThreadName="
                    + Thread.currentThread().getName() + " 首选进入！");
                for (int i = 0; i < Integer.MAX_VALUE; i++) {
                    String newString = new String();
                    Math.random();
                }
                semaphore.release();
            } else {
                System.out.println("ThreadName="
                    + Thread.currentThread().getName() + " 未成功进入！");
            }
            // 方法 release 对应的 permits 值也要更改
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的效果如图 1-20 所示。

更改 Service.java 类代码如下：

```
for (int i = 0; i < Integer.MAX_VALUE; i++) {
    // String newString = new String();
    // Math.random();
}
```

程序运行结果如图 1-21 所示。



图 1-20 线程 B 未获得许可

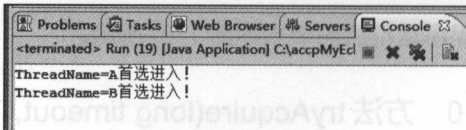


图 1-21 双双获得许可

1.1.11 方法 tryAcquire(int permits, long timeout, TimeUnit unit) 的使用

有参方法 tryAcquire(int permits, long timeout, TimeUnit unit) 的作用是在指定的时间内尝试地获得 x 个许可，如果获取不到则返回 false。

创建 Java 项目 Semaphore_tryAcquire_4，将项目 Semaphore_tryAcquire_3 中的源代码复制到 Semaphore_tryAcquire_4 中，更改类 Service.java 代码如下：

```
package service;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

public class Service {

    private Semaphore semaphore = new Semaphore(3);

    public void testMethod() {
        //1 改成 3
        try {
            if (semaphore.tryAcquire(3, 3, TimeUnit.SECONDS)) {
                System.out.println("ThreadName="
                    + Thread.currentThread().getName() + " 首选进入! ");
                for (int i = 0; i < Integer.MAX_VALUE; i++) {
                    String newString = new String();
                    Math.random();
                }
                // 方法 release 对应的 permits 值也要更改
                semaphore.release(3);
            } else {
                System.out.println("ThreadName="
                    + Thread.currentThread().getName() + " 未成功进入! ");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的效果如图 1-22 所示。

更改 Service.java 类代码如下：

```
for (int i = 0; i < Integer.MAX_VALUE; i++) {
}
```

程序运行结果如图 1-23 所示。

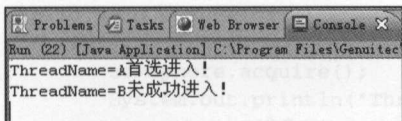


图 1-22 线程 B 未获得许可

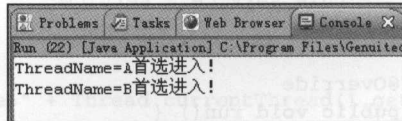


图 1-23 都获得许可

1.1.12 多进路 - 多处理 - 多出路实验

本实现的目标是允许多个线程同时处理任务，更具体来讲，也就是每个线程都在处理自己的任务。

创建实验用的项目 Semaphore_MoreToOne_1，类 Service.java 代码如下：

```
package service;

import java.util.concurrent.Semaphore;

public class Service {

    private Semaphore semaphore = new Semaphore(3);

    public void sayHello() {
        try {
            semaphore.acquire();
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                + " 准备");
            System.out.println("begin hello " + System.currentTimeMillis());
            for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName()
                    + " 打印 " + (i + 1));
            }
            System.out.println(" end hello " + System.currentTimeMillis());
            semaphore.release();
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                + " 结束");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

线程类 MyThread.java 代码如下：

```
package extthread;

import service.Service;

public class MyThread extends Thread {

    private Service service;

    public MyThread(Service service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.sayHello();
    }
}
```

运行类 Run.java 代码如下:

```
package test.run;

import service.Service;
import extthread.MyThread;

public class Run {

    public static void main(String[] args) {
        Service service = new Service();

        MyThread[] threadArray = new MyThread[12];
        for (int i = 0; i < threadArray.length; i++) {
            threadArray[i] = new MyThread(service);
            threadArray[i].start();
        }
    }
}
```

程序运行结果如图 1-24 所示。

运行的效果是多个线程同时进入,而多个线程又几乎同时执行完毕。

1.1.13 多进路-单处理-多出路实验

本实现的目标是允许多个线程同时处理任务,但执行任务的顺序却是同步的,也就是阻塞的,所以也称单处理。

创建实验用的项目 Semaphore_MoreToOne_2,将 Semaphore_MoreToOne_1 项目中的所有源代码复制到项目 Semaphore_MoreToOne_2 中,更改类 Service.java 代码如下:

```
package service;

import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

public class Service {

    private Semaphore semaphore = new Semaphore(3);
    private ReentrantLock lock = new ReentrantLock();

    public void sayHello() {
        try {
            semaphore.acquire();
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                + " 准备");
        }
    }
}
```

```

lock.lock();
System.out.println("begin hello " + System.currentTimeMillis());
for (int i = 0; i < 5; i++) {
    System.out.println(Thread.currentThread().getName() + " 打印 "
        + (i + 1));
}
System.out.println(" end hello " + System.currentTimeMillis());
lock.unlock();
semaphore.release();
System.out.println("ThreadName=" + Thread.currentThread().getName()
    + " 结束");
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

```

在代码中加入了 `ReentrantLock` 对象，保证了同步性。

程序运行结果如图 1-25 所示。

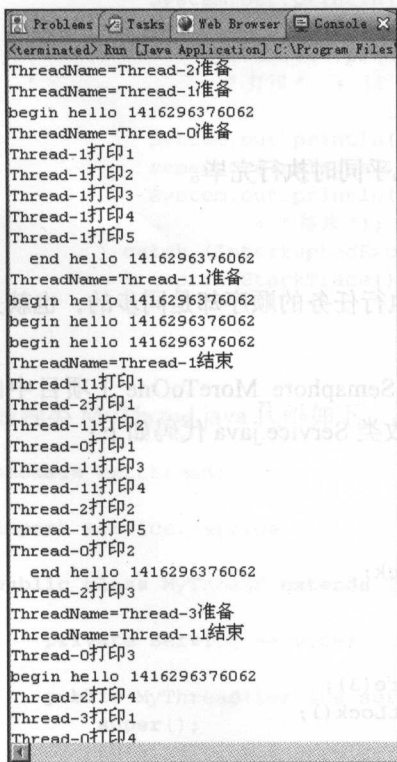


图 1-24 打印循环中的内容为乱序

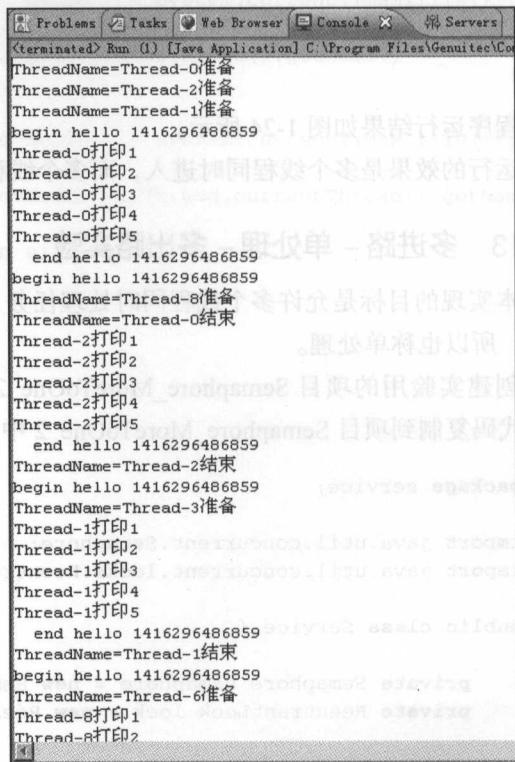


图 1-25 打印循环中的内容为有序

在单处理步骤中实现了同步，所以打印的效果呈 1, 2, 3, 4, 5 这样的顺序。

1.1.14 使用 Semaphore 创建字符串池

类 Semaphore 可以有效地对并发执行任务的线程数量进行限制, 这种功能可以应用在 pool 池技术中, 可以设置同时访问 pool 池中数据的线程数量。

本实验的功能是同时有若干个线程可以访问池中的数据, 但同时只有一个线程可以取得数据, 使用完后再放回池中。

创建测试用的项目 Semaphore_Pool_List, 类 ListPool.java 代码如下:

```
package tools;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class ListPool {

    private int poolMaxSize = 3;
    private int semaphorePermits = 5;
    private List<String> list = new ArrayList<String>();
    private Semaphore concurrencySemaphore = new Semaphore(semaphorePermits);
    private ReentrantLock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    public ListPool() {
        super();
        for (int i = 0; i < poolMaxSize; i++) {
            list.add("高洪岩" + (i + 1));
        }
    }

    public String get() {
        String getString = null;
        try {
            concurrencySemaphore.acquire();
            lock.lock();
            while (list.size() == 0) {
                condition.await();
            }
            getString = list.remove(0);
            lock.unlock();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return getString;
    }

    public void put(String stringValue) {
        lock.lock();
        list.add(stringValue);
        condition.signalAll();
        lock.unlock();
    }
}
```



```

        concurrencySemaphore.release();
    }
}

```

线程类 MyThread.java 代码如下：

```

package extthread;

import tools.ListPool;

public class MyThread extends Thread {

    private ListPool listPool;

    public MyThread(ListPool listPool) {
        super();
        this.listPool = listPool;
    }

    @Override
    public void run() {
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            String getString = listPool.get();
            System.out.println(Thread.currentThread().getName() + " 取得值 "
                + getString);
            listPool.put(getString);
        }
    }
}

```

运行类 Run.java 代码如下：

```

package test;

import tools.ListPool;
import extthread.MyThread;

public class Run {

    public static void main(String[] args) {

        ListPool pool = new ListPool();

        MyThread[] threadArray = new MyThread[12];
        for (int i = 0; i < threadArray.length; i++) {
            threadArray[i] = new MyThread(pool);
        }
        for (int i = 0; i < threadArray.length; i++) {
            threadArray[i].start();
        }
    }
}

```

程序运行结果如图 1-26 所示。



图 1-26 部分运行结果

为了验证更多的情况，更改类代码如下：

```
private int poolMaxSize = 5;
private int semaphorePermits = 3;
```

程序正常运行，不出现异常。

更改类代码如下：

```
private int poolMaxSize = 5;
private int semaphorePermits = 5;
```

程序正常运行，也不出现异常。

1.1.15 使用 Semaphore 实现多生产者 / 多消费者模式

本实验的目的不光是实现生产者与消费者模式，还要限制生产者与消费者的数量，这样代码的复杂性就提高一些，但好在使用 Semaphore 类实现这个功能还是比较简单的。

创建实验用的项目 repastTest，类 RepastService.java 代码如下：

```

package service;

import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class RepastService {

    volatile private Semaphore setSemaphore = new Semaphore(10); // 厨师
    volatile private Semaphore getSemaphore = new Semaphore(20); // 就餐者
    volatile private ReentrantLock lock = new ReentrantLock();
    volatile private Condition setCondition = lock.newCondition();
    volatile private Condition getCondition = lock.newCondition();
    // producePosition 变量的含义是最多只有 4 个盒子存放菜品
    volatile private Object[] producePosition = new Object[4];

    private boolean isEmpty() {
        boolean isEmpty = true;
        for (int i = 0; i < producePosition.length; i++) {
            if (producePosition[i] != null) {
                isEmpty = false;
                break;
            }
        }
        if (isEmpty == true) {
            return true;
        } else {
            return false;
        }
    }

    private boolean isFull() {
        boolean isFull = true;
        for (int i = 0; i < producePosition.length; i++) {
            if (producePosition[i] == null) {
                isFull = false;
                break;
            }
        }
        return isFull;
    }

    public void set() {
        try {
            // System.out.println("set");
            setSemaphore.acquire(); // 允许同时最多有 10 个厨师进行生产
            lock.lock();
            while (isFull()) {
                // System.out.println("生产者正在等待");
                setCondition.await();
            }
        }
    }
}

```

```

        for (int i = 0; i < producePosition.length; i++){
            if (producePosition[i] == null) {
                producePosition[i] = "数据 ";
                System.out.println(Thread.currentThread().getName()
                    + " 生产了 " + producePosition[i]);
                break;
            }
        }
        getCondition.signalAll();
        lock.unlock();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        setSemaphore.release();
    }
}

public void get() {
    try {
        //System.out.println("get");
        getSemaphore.acquire();    // 允许同时最多有 16 个就餐者
        lock.lock();
        while (isEmpty()) {
            //System.out.println("消费者在等待");
            getCondition.await();
        }
        for (int i = 0; i < producePosition.length; i++) {
            if (producePosition[i] != null) {
                System.out.println(Thread.currentThread().getName()
                    + " 消费了 " + producePosition[i]);
                producePosition[i] = null;
                break;
            }
        }
        setCondition.signalAll();
        lock.unlock();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        getSemaphore.release();
    }
}
}

```

两个线程类代码如图 1-27 所示。

运行类 Run.java 代码如下：

```

package test.run;

import service.RepastService;

```

```

import extthread.ThreadC;
import extthread.ThreadP;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        RepastService service = new RepastService();
        ThreadP[] arrayP = new ThreadP[60];
        ThreadC[] arrayC = new ThreadC[60];
        for (int i = 0; i < 60; i++) {
            arrayP[i] = new ThreadP(service);
            arrayC[i] = new ThreadC(service);
        }
        Thread.sleep(2000);
        for (int i = 0; i < 60; i++) {
            arrayP[i].start();
            arrayC[i].start();
        }
    }
}

```

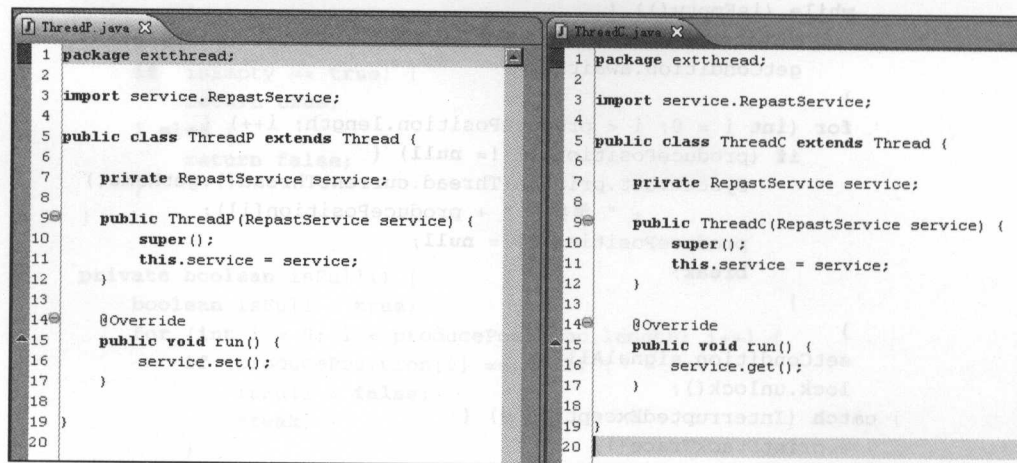


图 1-27 线程类代码

程序运行结果如下：

```

Thread-0 生产了 数据
Thread-3 消费了 数据
Thread-2 生产了 数据
Thread-8 生产了 数据
Thread-16 生产了 数据
Thread-12 生产了 数据
Thread-19 消费了 数据
Thread-11 消费了 数据

```

Thread-20 生产了 数据
 Thread-21 消费了 数据
 Thread-25 消费了 数据
 Thread-26 生产了 数据
 Thread-27 消费了 数据
 Thread-28 生产了 数据
 Thread-29 消费了 数据
 Thread-30 生产了 数据
 Thread-31 消费了 数据
 Thread-1 消费了 数据
 Thread-34 生产了 数据
 Thread-36 生产了 数据
 Thread-38 生产了 数据
 Thread-40 生产了 数据
 Thread-41 消费了 数据
 Thread-42 生产了 数据
 Thread-7 消费了 数据
 Thread-61 消费了 数据
 Thread-63 消费了 数据
 Thread-65 消费了 数据
 Thread-22 生产了 数据
 Thread-48 生产了 数据
 Thread-4 生产了 数据
 Thread-13 消费了 数据
 Thread-69 消费了 数据
 Thread-71 消费了 数据
 Thread-24 生产了 数据
 Thread-18 生产了 数据
 Thread-56 生产了 数据
 Thread-58 生产了 数据
 Thread-33 消费了 数据
 Thread-75 消费了 数据
 Thread-77 消费了 数据
 Thread-35 消费了 数据
 Thread-44 生产了 数据
 Thread-45 消费了 数据
 Thread-6 生产了 数据
 Thread-64 生产了 数据
 Thread-66 生产了 数据
 Thread-46 生产了 数据
 Thread-67 消费了 数据
 Thread-15 消费了 数据
 Thread-87 消费了 数据
 Thread-50 生产了 数据
 Thread-72 生产了 数据
 Thread-74 生产了 数据
 Thread-17 消费了 数据
 Thread-73 消费了 数据
 Thread-23 消费了 数据
 Thread-5 消费了 数据
 Thread-54 生产了 数据

Thread-60 生产了 数据
Thread-32 生产了 数据
Thread-82 生产了 数据
Thread-79 消费了 数据
Thread-37 消费了 数据
Thread-81 消费了 数据
Thread-39 消费了 数据
Thread-62 生产了 数据
Thread-83 消费了 数据
Thread-68 生产了 数据
Thread-10 生产了 数据
Thread-14 生产了 数据
Thread-92 生产了 数据
Thread-85 消费了 数据
Thread-89 消费了 数据
Thread-111 消费了 数据
Thread-113 消费了 数据
Thread-76 生产了 数据
Thread-52 生产了 数据
Thread-91 消费了 数据
Thread-93 消费了 数据
Thread-78 生产了 数据
Thread-80 生产了 数据
Thread-84 生产了 数据
Thread-99 消费了 数据
Thread-101 消费了 数据
Thread-103 消费了 数据
Thread-86 生产了 数据
Thread-47 消费了 数据
Thread-88 生产了 数据
Thread-108 生产了 数据
Thread-110 生产了 数据
Thread-112 生产了 数据
Thread-109 消费了 数据
Thread-115 消费了 数据
Thread-96 生产了 数据
Thread-98 生产了 数据
Thread-117 消费了 数据
Thread-119 消费了 数据
Thread-95 消费了 数据
Thread-97 消费了 数据
Thread-102 生产了 数据
Thread-100 生产了 数据
Thread-104 生产了 数据
Thread-105 消费了 数据
Thread-43 消费了 数据
Thread-106 生产了 数据
Thread-107 消费了 数据
Thread-49 消费了 数据
Thread-114 生产了 数据
Thread-90 生产了 数据
Thread-94 生产了 数据

```

Thread-70 生产了 数据
Thread-9 消费了 数据
Thread-51 消费了 数据
Thread-53 消费了 数据
Thread-55 消费了 数据
Thread-116 生产了 数据
Thread-118 生产了 数据
Thread-57 消费了 数据
Thread-59 消费了 数据

```

工具 editplus 中的行数如图 1-28 所示。

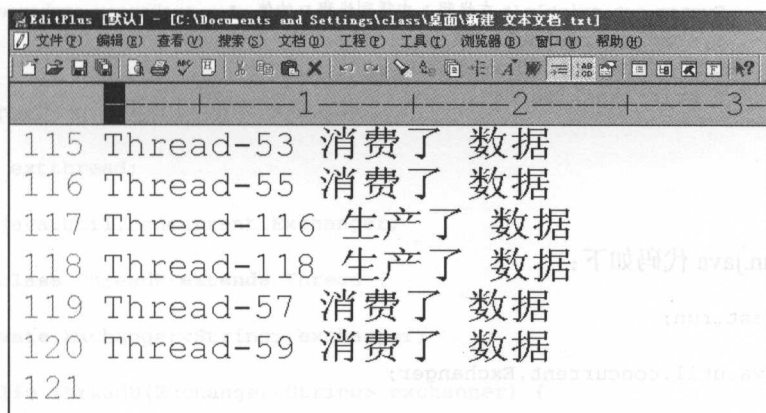


图 1-28 打印的行数

类 Semaphore 提供了限制并发线程数的功能，此功能在默认的 synchronized 中是不提供的。

1.2 Exchanger 的使用

类 Exchanger[iks'tʃeindʒə] 的功能可以使 2 个线程之间传输数据，它比生产者/消费者模式使用的 wait/notify 要更加方便。所以在本节将介绍使用此类在 2 个线程之间传递任意数据类型的数据，Exchanger 类的使用与结构相当简单，主要的学习点就是 exchange() 方法。

1.2.1 方法 exchange() 阻塞的特性

类 Exchanger 中的 exchange() 方法具有阻塞的特色，也就是此方法被调用后等待其他线程来取得数据，如果没有其他线程取得数据，则一直阻塞等待。

创建测试用的项目 Exchanger_1，创建类 ThreadA.java 代码如下：

```
package extthread;
```

```
import java.util.concurrent.Exchanger;
```

```

public class ThreadA extends Thread {
    private Exchanger<String> exchanger;

    public ThreadA(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        try {
            System.out.println("在线程 A 中得到线程 B 的值 =" + exchanger.exchange(" 中国人 A"));
            System.out.println("A end!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

运行类 Run.java 代码如下：

```

package test.run;

import java.util.concurrent.Exchanger;

import extthread.ThreadA;

public class Run {
    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<String>();
        ThreadA a = new ThreadA(exchanger);
        a.start();
        System.out.println("main end!");
    }
}

```

程序运行效果如图 1-29 所示。

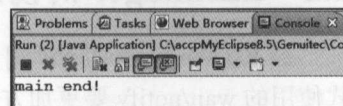


图 1-29 出现阻塞效果

1.2.2 方法 exchange() 传递数据

创建测试用的项目 Exchanger_2，创建 ThreadA.java 类代码如下：

```

package extthread;

import java.util.concurrent.Exchanger;

public class ThreadA extends Thread {

    private Exchanger<String> exchanger;

```

```

public ThreadA(Exchanger<String> exchanger) {
    super();
    this.exchanger = exchanger;
}

@Override
public void run() {
    try {
        System.out.println("在线程 A 中得到线程 B 的值 =" + exchanger.exchange("中国人 A"));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

创建 ThreadB.java 类代码如下:

```

package extthread;

import java.util.concurrent.Exchanger;

public class ThreadB extends Thread {

    private Exchanger<String> exchanger;

    public ThreadB(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        try {
            System.out.println("在线程 B 中得到线程 A 的值 =" + exchanger.exchange("中国人 B"));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
}

```

运行类 Run.java 代码如下:

```

package test.run;

import java.util.concurrent.Exchanger;

import extthread.ThreadA;
import extthread.ThreadB;

public class Run {
    public static void main(String[] args) {

```

```

    Exchanger<String> exchanger = new Exchanger<String>();
    ThreadA a = new ThreadA(exchanger);
    ThreadB b = new ThreadB(exchanger);
    a.start();
    b.start();
}
}

```

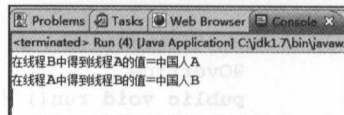


图 1-30 成功交换数据

程序运行后的效果如图 1-30 所示。

1.2.3 方法 `exchange(V x, long timeout, TimeUnit unit)` 与超时

当调用 `exchange(V x, long timeout, TimeUnit unit)` 方法后在指定的时间内没有其他线程获取数据，则出现超时异常。

创建测试用的项目 `Exchanger_3`，创建类 `ThreadA.java` 代码如下：

```

package extthread;

import java.util.concurrent.Exchanger;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class ThreadA extends Thread {

    private Exchanger<String> exchanger;

    public ThreadA(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        try {
            System.out.println(" 在线程 A 中得到线程 B 的值 = "
                + exchanger.exchange(" 中国人 A", 5, TimeUnit.SECONDS));
            System.out.println("A end!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (TimeoutException e) {
            //TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

运行类 `Run.java` 代码如下：

```

package test.run;

```

```
import java.util.concurrent.Exchanger;

import extthread.ThreadA;

public class Run {
    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<String>();
        ThreadA a = new ThreadA(exchanger);
        a.start();
        System.out.println("main end!");
    }
}
```

程序运行效果如图 1-31 所示。

```
main end!
java.util.concurrent.TimeoutException
    at java.util.concurrent.Exchanger.exchange(Exchanger.java:683)
    at extthread.ThreadA.run(ThreadA.java:20)
```

图 1-31 出现超时异常

1.3 本章总结

类 Semaphore 的主要作用是限制并发执行的线程个数，它具有 synchronized 所不具有的强大功能，比如等待获得许可的同时可以加入等待时间，还有尝试是否可以持有锁等这类扩展功能，可以说 Semaphore 类是强有力的控制并发线程个数的解决方案之一，而 Exchanger 是线程间传输数据的方式之一，而且在传输的数据类型上并没有任何限制。

CountDownLatch 和 CyclicBarrier 的使用

在 Java 并发包中控制线程的同步还有一些比较常见的工具类 `CountDownLatch` 和 `CyclicBarrier`，这两个工具类可以使线程在同步的处理上更加灵活，比如支持同步计数重置、等待同步线程个数等常见功能。

那么在本章就主要介绍 `CountDownLatch` 和 `CyclicBarrier` 这两个工具类的使用，详细到每个类的 API 的具体使用与应用场景，这两个工具将同步与线程“组团”做任务完美进行了支持。

2.1 CountDownLatch 的使用

单词 `Latch` 的发音为 [lætʃ]，中文翻译是门闩，也就是有“门锁”的功能，所以当门没有打开时， N 个人是不能进入屋内的，也就是 N 个线程是不能继续向下运行的，支持这样的特性可以控制线程执行任务的时机，使线程以“组团”的方式一起执行任务。

类 `CountDownLatch` 所提供的功能是判断 `count` 计数不为 0 时则当前线程呈 `wait` 状态，也就是在屏障处等待，该类的全部方法列表如图 2-1 所示。

类 `CountDownLatch` 也是一个同步功能的辅助类，使用效果是给定一个计数，当使用这个 `CountDownLatch` 类的线程判断计数不为 0 时，则呈 `wait` 状态，如果为 0 时则继续运行。

实现等待与继续运行的效果分别需要使用 `await()` 和 `countDown()` 方法来进行。调用 `await()` 方法时判断计数是否为 0，如果不为 0 则呈等待状态。其他线程可以调用 `countDown()` 方法将计数减 1，当计数减到为 0 时，呈等待的线程继续运行。而方法 `getCount()` 就是获得当前的计数个数。

要说明的是，计数无法被重置，如果需要重置计数，请考虑使用 `CyclicBarrier` 类。

类 `CountDownLatch` 的计数是减法操作。

```

● await() : void - CountdownLatch
● await(long timeout, TimeUnit unit) : boolean - CountdownLatch
● countDown() : void - CountdownLatch
● equals(Object obj) : boolean - Object
● getClass() : Class<?> - Object
● getCount() : long - CountdownLatch
● hashCode() : int - Object
● notify() : void - Object
● notifyAll() : void - Object
● toString() : String - CountdownLatch
● wait() : void - Object
● wait(long timeout) : void - Object
● wait(long timeout, int nanos) : void - Object

```

图 2-1 类 Count Down Latch 的全部方法列表

2.1.1 初步使用

在本节将实现一个当 CountdownLatch 的计数不为 0 时，线程呈阻塞状态的效果。通过本实验可以看到线程执行任务具有“组团”的特性了。

创建测试用的项目 CountdownLatch_test1，类 MyService.java 代码如下：

```

package service;

import java.util.concurrent.CountDownLatch;

public class MyService {

    private CountDownLatch down = new CountDownLatch(1);

    public void testMethod() {
        try {
            System.out.println("A");
            down.await();
            System.out.println("B");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void downMethod() {
        System.out.println("X");
        down.countDown();
    }
}

```

代码 new CountdownLatch(1) 的作用是创建 1 个计数的 CountdownLatch 类的对象，当线程执行 down.await() 代码时呈等待状态，程序不向下继续运行。程序执行 down.countDown() 代码时计数由 1 变成 0，以前呈等待状态的线程继续向下运行。

线程类 MyThread.java 代码如下：

```

package extthread;

```

```
import service.MyService;

public class MyThread extends Thread {

    private MyService myService;

    public MyThread(MyService myService) {
        super();
        this.myService = myService;
    }

    @Override
    public void run() {
        myService.testMethod();
    }

}
```

运行类 Run.java 代码如下：

```
package test.run;

import service.MyService;
import extthread.MyThread;

public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        MyThread t = new MyThread(service);
        t.start();
        Thread.sleep(2000);
        service.downMethod();
    }
}
```

程序运行结果如图 2-2 所示。

await() 方法还有重载的写法 await(long timeout, TimeUnit unit)，此方法在 2.1.5 节有测试的代码。

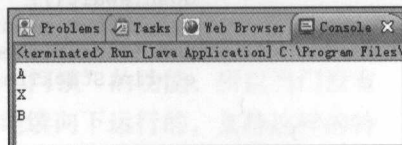


图 2-2 运行结果

2.1.2 裁判在等全部的运动员到来

本实验想要实现的是裁判员要等待所有运动员全部到来的效果，这个实验比前面章节的实验更加复杂，主要论证的就是多个线程与同步点间阻塞的特性，线程都必须到达同步点后才可以继续向下运行。

创建测试用的项目 CountdownLatch_test3，类 MyService.java 代码如下：

线程类 MyThread.java 代码如下：

```
package extthread;

import java.util.concurrent.CountDownLatch;
```

```

public class MyThread extends Thread {

    private CountdownLatch maxRunner;

    public MyThread(CountDownLatch maxRunner) {
        super();
        this.maxRunner = maxRunner;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(2000);
            maxRunner.countDown();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

运行类 Run.java 代码如下:

```

package test.run;

import java.util.concurrent.CountDownLatch;
import extthread.MyThread;

public class Run {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch maxRunner = new CountDownLatch(10);
        MyThread[] tArray = new MyThread[Integer.parseInt("10")
            + maxRunner.getCount()];
        for (int i = 0; i < tArray.length; i++) {
            tArray[i] = new MyThread(maxRunner);
            tArray[i].setName("线程 " + (i + 1));
            tArray[i].start();
        }
        maxRunner.await();
        System.out.println("都回来了!");
    }
}

```

程序运行结果如图 2-3 所示。

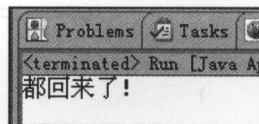


图 2-3 大家都到了

2.1.3 各就各位准备比赛

本实验想要实现的是裁判员要等待所有运动员各就各位后全部准备完毕, 再开始比赛的效果。

创建测试用的项目 CountdownLatch_test2, 类 MyService.java 代码如下:

```

package service;

import java.util.concurrent.CountDownLatch;

public class MyService {

    private CountDownLatch down = new CountDownLatch(1);

    public void testMethod() {
        try {
            System.out.println(Thread.currentThread().getName() + " 准备");
            down.await();
            System.out.println(Thread.currentThread().getName() + " 结束");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void downMethod() {
        System.out.println(" 开始");
        down.countDown();
    }
}

```

线程类 MyThread.java 代码如下：

```

package extthread;

import service.MyService;

public class MyThread extends Thread {

    private MyService myService;

    public MyThread(MyService myService) {
        super();
        this.myService = myService;
    }

    @Override
    public void run() {
        myService.testMethod();
    }
}

```

运行类 Run.java 代码如下：

```

package test.run;

import service.MyService;
import extthread.MyThread;

```

```

public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        MyThread[] tArray = new MyThread[10];
        for (int i = 0; i < tArray.length; i++) {
            tArray[i] = new MyThread(service);
            tArray[i].setName("线程" + (i + 1));
            tArray[i].start();
        }
        Thread.sleep(2000);
        service.downMethod();
    }
}

```

程序运行结果如图 2-4 所示。

说明一下，此实验虽然运行成功，但并不能保证在 main 主线程中执行了 service.downMethod(); 时，所有的工作线程都呈 wait 状态，因为某些线程有可能准备的时间花费较长，可能耗用的时间超过 2 秒，这时如果在第 2 秒时调用 service.downMethod(); 方法就达不到“唤醒所有线程”继续向下运行的目的了，也就是说裁判员没有等全部的运动员到来时，就让发令枪响起开始比赛了，这是不对的，所以需要我对代码进行修改，来达到相对比较完善的比赛流程，下面的章节将实现这个案例。

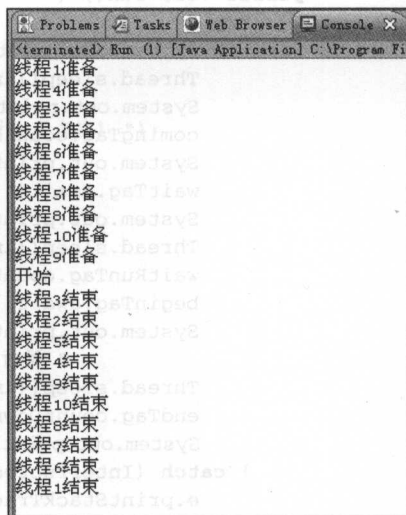


图 2-4 运行结果

2.1.4 完整的比赛流程

本节要使用 CountdownLatch 类来实现“所有的线程”呈 wait 后再统一唤醒的效果，此实验大量使用 CountdownLatch 类来实现业务要求的同步效果。

创建实验用的项目，名称为 CountdownLatch_test2_3_ext，创建 MyThread.java 类代码如下：

```

package extthread;

import java.util.concurrent.CountDownLatch;

public class MyThread extends Thread {

    private CountDownLatch comingTag; // 裁判等待所有运动员到来
    private CountDownLatch waitTag; // 等待裁判说准备开始
    private CountDownLatch waitRunTag; // 等待起跑
    private CountDownLatch beginTag; // 起跑
    private CountDownLatch endTag; // 所有运动员到达终点

    public MyThread(CountDownLatch comingTag, CountDownLatch waitTag,

```



```

        CountdownLatch waitRunTag, CountdownLatch beginTag,
        CountdownLatch endTag) {
    super();
    this.comingTag = comingTag;
    this.waitTag = waitTag;
    this.waitRunTag = waitRunTag;
    this.beginTag = beginTag;
    this.endTag = endTag;
}

@Override
public void run() {
    try {
        System.out.println(" 运动员使用不同交通工具不同速度到达起跑点，正向这头走！");
        Thread.sleep((int) (Math.random() * 10000));
        System.out.println(Thread.currentThread().getName() + " 到起跑点了！");
        comingTag.countDown();
        System.out.println(" 等待裁判说准备！");
        waitTag.await();
        System.out.println(" 各就各位！准备起跑姿势！");
        Thread.sleep((int) (Math.random() * 10000));
        waitRunTag.countDown();
        beginTag.await();
        System.out.println(Thread.currentThread().getName()
            + " 运行员起跑 并且跑赛过程用时不确定");
        Thread.sleep((int) (Math.random() * 10000));
        endTag.countDown();
        System.out.println(Thread.currentThread().getName() + " 运行员到达终点");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

运行类 Run.java 代码如下：

```

package test;

import java.util.concurrent.CountDownLatch;
import extthread.MyThread;

public class Run {

    public static void main(String[] args) {
        try {
            CountdownLatch comingTag = new CountdownLatch(10);
            CountdownLatch waitTag = new CountdownLatch(1);
            CountdownLatch waitRunTag = new CountdownLatch(10);
            CountdownLatch beginTag = new CountdownLatch(1);
            CountdownLatch endTag = new CountdownLatch(10);

            MyThread[] threadArray = new MyThread[10];

```

```

    for (int i = 0; i < threadArray.length; i++) {
        threadArray[i] = new MyThread(comingTag, waitTag, waitRunTag,
            beginTag, endTag);
        threadArray[i].start();
    }
    System.out.println(" 裁判员在等待选手的到来! ");
    comingTag.await();
    System.out.println(" 裁判看到所有运动员来了, 各就各位前“巡视”用时 5 秒");
    Thread.sleep(5000);
    waitTag.countDown();
    System.out.println(" 各就各位! ");
    waitRunTag.await();
    Thread.sleep(2000);
    System.out.println(" 发令枪响起! ");
    beginTag.countDown();
    endTag.await();
    System.out.println(" 所有运动员到达, 统计比赛名次! ");
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

```

程序运行结果如下所示:

```

运动员使用不同交通工具不同速度到达起跑点, 正向这头走!
运动员使用不同交通工具不同速度到达起跑点, 正向这头走!
运动员使用不同交通工具不同速度到达起跑点, 正向这头走!
运动员使用不同交通工具不同速度到达起跑点, 正向这头走!
运动员使用不同交通工具不同速度到达起跑点, 正向这头走!
运动员使用不同交通工具不同速度到达起跑点, 正向这头走!
运动员使用不同交通工具不同速度到达起跑点, 正向这头走!
运动员使用不同交通工具不同速度到达起跑点, 正向这头走!
裁判员在等待选手的到来!
运动员使用不同交通工具不同速度到达起跑点, 正向这头走!
Thread-9 到起跑点了!
等待裁判说准备!
Thread-2 到起跑点了!
等待裁判说准备!
Thread-7 到起跑点了!
等待裁判说准备!
Thread-5 到起跑点了!
等待裁判说准备!
Thread-4 到起跑点了!
等待裁判说准备!
Thread-0 到起跑点了!
等待裁判说准备!
Thread-1 到起跑点了!
等待裁判说准备!
Thread-3 到起跑点了!

```

Thread-8 到起跑点了!

Thread-6 到起跑点了!

裁判看到所有运动员来了，各就各位前“巡视”用时5秒

各就各位！准备起跑姿势！

各就各位！准备起跑姿势！

各就各位！准备起跑姿势！

各就各位！准备起跑姿势！

各就各位！准备起跑姿势！

发令枪响起！

Thread-1 运行员起跑 并且跑赛过程用时不确定

Thread-7 运行员起跑 并且跑赛过程用时不确定

Thread-5 运行员起跑 并且跑赛过程用时不确定

Thread-2 运行员起跑 并且跑赛过程用时不确定

Thread-6 运行员起跑 并且跑赛过程用时不确定

Thread-8 运行员到达终点

Thread-5 运行员到达终点

Thread-7 运行员到达终点

Thread-3 运行员到达终点

Thread-1 运行员到达终点

2.1.5

创建

```
import java.util.concurrent.TimeUnit;
```

```

public class MyService {

    public CountdownLatch count = new CountdownLatch(1);

    public void testMethod() {
        try {
            System.out.println(Thread.currentThread().getName() + " 准备 "
                + System.currentTimeMillis());
            count.await(3, TimeUnit.SECONDS);
            System.out.println(Thread.currentThread().getName() + " 结束 "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

线程类 MyThread.java 代码如下:

```

package extthread;

import service.MyService;

public class MyThread extends Thread {

    private MyService service;

    public MyThread(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.testMethod();
    }

}

```

运行类 Run.java 代码如下:

```

package test.run;

import service.MyService;
import extthread.MyThread;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        MyThread[] threadArray = new MyThread[3];
        for (int i = 0; i < threadArray.length; i++) {

```

```

        threadArray[i] = new MyThread(service);
    }
    for (int i = 0; i < threadArray.length; i++) {
        threadArray[i].start();
    }
}

```

程序运行结果如图 2-5 所示。

实现了 3 秒后继续向下运行的效果。

```

<terminated> Run [Java Application] C:\
Thread-0 准备 1430203721593
Thread-2 准备 1430203721593
Thread-1 准备 1430203721593
Thread-1 结束 1430203724593
Thread-2 结束 1430203724593
Thread-0 结束 1430203724593

```

图 2-5 运行结果

2.1.6 方法 getCount() 的使用

方法 getCount() 获取当前计数的值。

创建名称为 CountdownLatch_getCount 的 Java 项目，创建类 Run.java 代码如下：

```

package test.run;

import java.util.concurrent.CountDownLatch;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        CountdownLatch count = new CountdownLatch(3);
        System.out.println("main getCount1=" + count.getCount());
        count.countDown();
        System.out.println("main getCount2=" + count.getCount());
        count.countDown();
        System.out.println("main getCount3=" + count.getCount());
        count.countDown();
        System.out.println("main getCount4=" + count.getCount());
        count.countDown();
        System.out.println("main getCount5=" + count.getCount());
        count.countDown();
        System.out.println("main getCount6=" + count.getCount());
    }
}

```

```

<terminated> Run (1) [jav
main getCount1=3
main getCount2=2
main getCount3=1
main getCount4=0
main getCount5=0
main getCount6=0

```

程序运行结果如图 2-6 所示。

图 2-6 运行结果

2.2 CyclicBarrier 的使用

单词 Cyclic 的发音为 ['saiklik,'siklik]，是周期、循环的意思。单词 Barrier 的发音为 ['bæriə]，是门栅、关卡和障碍的意思。

类 CyclicBarrier 不仅有 CountdownLatch 所具有的功能，还可以实现屏障等待的功能，也

就是阶段性同步，它在使用上的意义在于可以循环地实现线程要一起做任务的目标，而不是像类 CountdownLatch 一样，仅仅支持一次线程与同步点阻塞的特性，该类的全部方法列表如图 2-7 所示。

类 CyclicBarrier 和 Semaphore 及 CountdownLatch 一样，也是一个同步辅助类。它允许一组线程互相等待，直到到达某个公共屏障点（common barrier point），这些线程必须实时地互相等待，这种情况下就可以使用 CyclicBarrier 类来方便地实现这样的功能。CyclicBarrier 类的公共屏障点可以重用，所以类的名称中有“cyclic 循环”的单词。

```

● await() : int - CyclicBarrier
● await(long timeout, TimeUnit unit) : int - CyclicBarrier
● equals(Object obj) : boolean - Object
● getClass() : Class<?> - Object
● getNumberWaiting() : int - CyclicBarrier
● getParties() : int - CyclicBarrier
● hashCode() : int - Object
● isBroken() : boolean - CyclicBarrier
● notify() : void - Object
● notifyAll() : void - Object
● reset() : void - CyclicBarrier
● toString() : String - Object
● wait() : void - Object
● wait(long timeout) : void - Object
● wait(long timeout, int nanos) : void - Object

```

图 2-7 类 Cyclic Barrier 的全部方法列表

通过上面段落中的文字可以发现，CyclicBarrier 类与 CountdownLatch 类在功能上有些相似，但在细节上还是有一些区别。

1) CountdownLatch 作用：一个线程或者多个线程，等待另外一个线程或多个线程完成某个事情之后才能继续执行。

2) CyclicBarrier 的作用：多个线程之间相互等待，任何一个线程完成之前，所有的线程都必须等待，所以对于 CyclicBarrier 来说，重点是“多个线程之间”任何一个线程没有完成任务，则所有的线程都必须等待。

类 CountdownLatch 和 CyclicBarrier 都有等待的功能，可以使用图示来解释它们之间的区别，如图 2-8 所示。

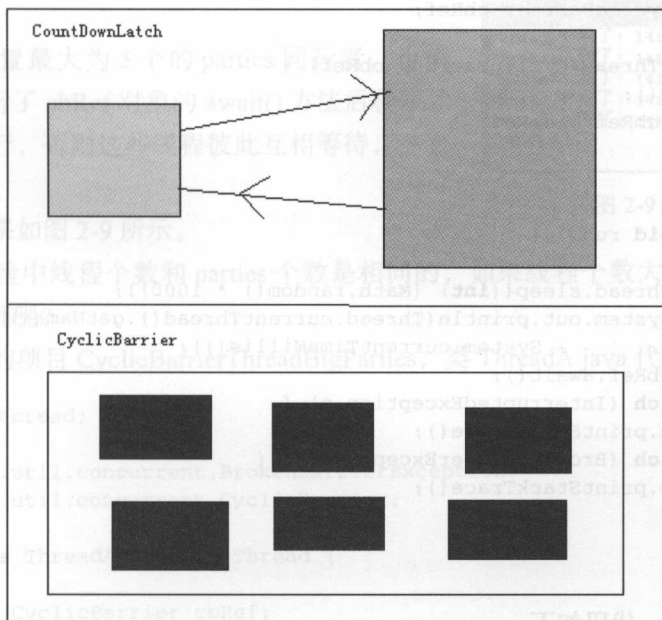


图 2-8 两个类之间的区别简图

从图 2-8 中来看，CountDownLatch 类的使用情况是两个角色之间互相等待，而 CyclicBarrier 的使用情况是同类互相等待。

和 CountDownLatch 类不同，类 CyclicBarrier 的计数是加法操作。

2.2.1 初步使用

在 CountDownLatch_test2_3_ext 项目中演示运动员比赛，实现准备、起跑以及到达终点的过程使用了 N 个 CountDownLatch 对象，从程序的设计结构上来看，是比较复杂并且繁琐的，后期在功能上的扩展是非常不方便的，最为主要的弊端是 CountDownLatch 类的计数不可以重置，想要再次获得同步的功能只有通过添加代码，增加代码的复杂度才可以换取要实现的功能。

在处理某些特殊业务的要求时，通常可以将 CountDownLatch 和 CyclicBarrier 类联合使用。

创建测试用的项目 CyclicBarrierBegin，本实验将要实现所有线程都到达同步点时再继续运行的效果，创建类 MyThread.java 代码如下：

```
package extthread;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class MyThread extends Thread {

    private CyclicBarrier cbRef;

    public MyThread(CyclicBarrier cbRef) {
        super();
        this.cbRef = cbRef;
    }

    @Override
    public void run() {
        try {
            Thread.sleep((int) (Math.random() * 1000));
            System.out.println(Thread.currentThread().getName() + " 到了！ "
                + System.currentTimeMillis());
            cbRef.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

运行类 Run.java 代码如下：

```

package test.run;

import java.util.concurrent.CyclicBarrier;

import extthread.MyThread;

public class Run {

    public static void main(String[] args) {
        CyclicBarrier cbRef = new CyclicBarrier(5, new Runnable() {
            public void run() {
                System.out.println("全都到了!");
            }
        });
        MyThread[] threadArray = new MyThread[5];
        for (int i = 0; i < threadArray.length; i++) {
            threadArray[i] = new MyThread(cbRef);
        }
        for (int i = 0; i < threadArray.length; i++) {
            threadArray[i].start();
        }
    }
}

```

程序代码:

```

CyclicBarrier cbRef = new CyclicBarrier(5, new Runnable() {
    public void run() {
        System.out.println("全都到了!");
    }
});

```

的作用是设置最大为 5 个的 parties 同行者, 也就是 5 个线程都执行了 cbRef 对象的 await() 方法后程序才可以继续向下运行, 否则这些线程彼此互相等待, 一直呈阻塞状态。

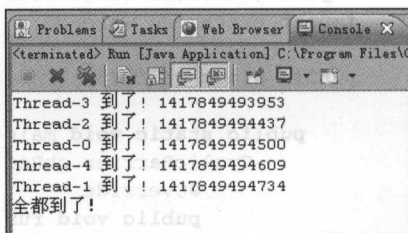


图 2-9 运行结果

程序运行结果如图 2-9 所示。

在上一个实验中线程个数和 parties 个数是相同的, 如果线程个数大于 parties 数量时能不能分批进行处理呢?

创建测试用的项目 CyclicBarrierThreadBigParties, 类 ThreadA.java 代码如下:

```

package extthread;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class ThreadA extends Thread {

    private CyclicBarrier cbRef;
}

```

```

public ThreadA(CyclicBarrier cbRef) {
    super();
    this.cbRef = cbRef;
}

@Override
public void run() {
    try {
        System.out.println(Thread.currentThread().getName() + " begin = "
            + System.currentTimeMillis() + " 等待凑齐 2 个继续运行");
        cbRef.await();
        System.out.println(Thread.currentThread().getName() + " end = "
            + System.currentTimeMillis() + " 已经凑齐 2 个继续运行");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        e.printStackTrace();
    }
}

```

类 Test.java 代码如下：

```

package test;

import java.util.concurrent.CyclicBarrier;
import extthread.ThreadA;

public class Test {

    public static void main(String[] args) throws InterruptedException {
        CyclicBarrier cbRef = new CyclicBarrier(2, new Runnable() {
            @Override
            public void run() {
                System.out.println("全来了！");
            }
        });

        for (int i = 0; i < 4; i++) {
            ThreadA threadA1 = new ThreadA(cbRef);
            threadA1.start();
            Thread.sleep(2000);
        }
    }
}

```

程序运行结果如下：

```

Thread-0 begin =1433640770453 等待凑齐 2 个继续运行
Thread-1 begin =1433640772453 等待凑齐 2 个继续运行
全来了！

```

```

Thread-1    end =1433640772453 已经凑齐 2 个继续运行
Thread-0    end =1433640772453 已经凑齐 2 个继续运行
Thread-2    begin =1433640774453 等待凑齐 2 个继续运行
Thread-3    begin =1433640776453 等待凑齐 2 个继续运行
全来了!
Thread-3    end =1433640776453 已经凑齐 2 个继续运行
Thread-2    end =1433640776453 已经凑齐 2 个继续运行

```

从运行结果来看,是可以实现分批进行比赛的效果,也就是每出现 2 个运动员就开始比赛。

2.2.2 验证屏障重置性及 getNumberWaiting() 方法的使用

类 CyclicBarrier 具有屏障重置性,在本节将验证这个特性。

本节将要结合方法 getNumberWaiting() 进行实验,该方法的作用是获得有几个线程已经到达屏障点。

创建名称为 CyclicBarrier_awaitAfterReset0 的 Java 项目,创建 ThreadA.java 类代码如下:

```

package extthread;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class ThreadA extends Thread {

    private CyclicBarrier cbRef;

    public ThreadA(CyclicBarrier cbRef) {
        super();
        this.cbRef = cbRef;
    }

    @Override
    public void run() {
        try {
            cbRef.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}

```

运行类 Test.java 代码如下:

```

package test;

import java.util.concurrent.CyclicBarrier;

```

```

import extthread.ThreadA;

public class Test {

    public static void main(String[] args) throws InterruptedException {
        CyclicBarrier cbRef = new CyclicBarrier(3);

        ThreadA threadA1 = new ThreadA(cbRef);
        threadA1.start();
        Thread.sleep(500);
        System.out.println(cbRef.getNumberWaiting());

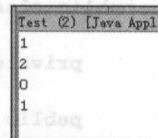
        ThreadA threadA2 = new ThreadA(cbRef);
        threadA2.start();
        Thread.sleep(500);
        System.out.println(cbRef.getNumberWaiting());

        ThreadA threadA3 = new ThreadA(cbRef);
        threadA3.start();
        Thread.sleep(500);
        System.out.println(cbRef.getNumberWaiting());

        ThreadA threadA4 = new ThreadA(cbRef);
        threadA4.start();
        Thread.sleep(500);
        System.out.println(cbRef.getNumberWaiting());
    }
}

```

程序运行后的效果如图 2-10 所示。



The screenshot shows a small Java window with the title 'Test (2) [Java Appl...]' and a text area containing the following output:

```

1
2
0
1

```

从运行结果来看，类 `CyclicBarrier` 的确具有屏障重置性，也就是 图 2-10 运行结果 `parties` 的值可以重置归 0。

2.2.3 用 `CyclicBarrier` 类实现阶段跑步比赛

2.2.2 节已经验证 `CyclicBarrier` 类具有屏障重置性，在本节中将丰富跑步比赛这个案例的内容，借助 `CyclicBarrier` 类具有计数重置性实现多赛段的比赛实验。

创建测试用的项目 `CyclicBarrier_run1`，创建类 `MyService.java` 代码如下：

```

package service;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class MyService {
    private CyclicBarrier cbRef;

    public MyService(CyclicBarrier cbRef) {
        super();
        this.cbRef = cbRef;
    }
}

```

```

public void beginRun() {
    try {
        long sleepValue = (int) (Math.random() * 10000);
        Thread.sleep(sleepValue);
        System.out.println(Thread.currentThread().getName() + " "
            + System.currentTimeMillis() + " begin 跑第 1 阶段 "
            + (cbRef.getNumberWaiting() + 1));
        cbRef.await();
        System.out.println(Thread.currentThread().getName() + " "
            + System.currentTimeMillis() + " end 跑第 1 阶段 " + " "
            + cbRef.getNumberWaiting());
        // =====
        sleepValue = (int) (Math.random() * 10000);
        Thread.sleep(sleepValue);
        System.out.println(Thread.currentThread().getName() + " "
            + System.currentTimeMillis() + " begin 跑第 2 阶段 "
            + (cbRef.getNumberWaiting() + 1));
        cbRef.await();
        System.out.println(Thread.currentThread().getName() + " "
            + System.currentTimeMillis() + " end 跑第 2 阶段 " + " "
            + cbRef.getNumberWaiting());
    } catch (InterruptedException e) {
        //TODO Auto-generated catch block
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        //TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}

```

创建类 ThreadA.java 代码如下:

```

package extthread;

import service.MyService;

public class ThreadA extends Thread {

    private MyService service;

    public ThreadA(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.beginRun();
    }
}

```


创建类 Test.java 代码如下：

```
package test;

import java.util.concurrent.CyclicBarrier;

import service.MyService;
import extthread.ThreadA;

public class Test {

    public static void main(String[] args) throws InterruptedException {
        CyclicBarrier cbRef = new CyclicBarrier(2);

        MyService service = new MyService(cbRef);

        ThreadA threadA1 = new ThreadA(service);
        threadA1.setName("A");
        threadA1.start();

        ThreadA threadA2 = new ThreadA(service);
        threadA2.setName("B");
        threadA2.start();

        ThreadA threadA3 = new ThreadA(service);
        threadA3.setName("C");
        threadA3.start();

        ThreadA threadA4 = new ThreadA(service);
        threadA4.setName("D");
        threadA4.start();
    }
}
```

程序运行结果如下：

```
A 1433641140468 begin 跑第 1 阶段 1
D 1433641140968 begin 跑第 1 阶段 2
D 1433641140968 end 跑第 1 阶段 0
A 1433641140968 end 跑第 1 阶段 0
D 1433641142296 begin 跑第 2 阶段 1
A 1433641142453 begin 跑第 2 阶段 2
A 1433641142453 end 跑第 2 阶段 0
D 1433641142453 end 跑第 2 阶段 0
C 1433641143171 begin 跑第 1 阶段 1
B 1433641146281 begin 跑第 1 阶段 2
B 1433641146281 end 跑第 1 阶段 0
C 1433641146281 end 跑第 1 阶段 0
B 1433641150937 begin 跑第 2 阶段 1
C 1433641151265 begin 跑第 2 阶段 2
C 1433641151265 end 跑第 2 阶段 0
B 1433641151265 end 跑第 2 阶段 0
```

此实验说明 CyclicBarrier 类的 parties 值从 1 到 2, 然后再恢复成 0 的过程, 证明 CyclicBarrier 类的屏障点是可以复用的。

其实线程 ABCD 每到达一个屏障点时的组合有可能是随机的, 有可能 A 和 B 到达第一屏障, 而到达第二屏障时的组合却是 A 和 D, 因为线程 A 和 D 的 sleep(long) 值较小, 也就是 sleep(long) 用时最少的互相组合, 继续向下一个屏障行进。

2.2.4 方法 isBroken() 的使用

方法 isBroken() 查询此屏障是否处于损坏状态。

创建项目 CyclicBarrier_run2, 创建类 MyService.java 代码如下:

```
package service;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class MyService {

    private CyclicBarrier cbRef;

    public MyService(CyclicBarrier cbRef) {
        super();
        this.cbRef = cbRef;
    }

    private void beginRun(int count) {
        try {
            System.out.println(Thread.currentThread().getName()
                + " 到了 在等待其他人都到了开始起跑 ");
            if (Thread.currentThread().getName().equals("Thread-2")) {
                System.out.println("thread-2 进来了");
                Thread.sleep(5000);
                Integer.parseInt("a");
            }
            cbRef.await();
            System.out.println("都到了, 开始跑!");
            System.out.println(Thread.currentThread().getName() + " 到达终点, 并结束第 "
                + count + " 赛段 ");
        } catch (InterruptedException e) {
            System.out.println("进入了 InterruptedException e " + cbRef.isBroken());
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            System.out.println("进入了 BrokenBarrierException e "
                + cbRef.isBroken());
            e.printStackTrace();
        }
    }

    public void testA() {
```

```

// 比赛 1 个赛段
for (int i = 0; i < 1; i++) {
    beginRun(i + 1);
}
}
}

```

线程类 `MyThread.java` 代码如下：

```

package extthread;

import service.MyService;

public class MyThread extends Thread {

    private MyService myService;

    public MyThread(MyService myService) {
        super();
        this.myService = myService;
    }

    @Override
    public void run() {
        myService.testA();
    }

}

```

运行类 `Run.java` 代码如下：

```

package test;

import java.util.concurrent.CyclicBarrier;

import service.MyService;
import extthread.MyThread;

public class Run {

    public static void main(String[] args) {
        int parties = 4;
        CyclicBarrier cbRef = new CyclicBarrier(parties, new Runnable() {
            public void run() {
                System.out.println(" 都到了！ ");
            }
        });

        MyService myService = new MyService(cbRef);

        MyThread[] threadArray = new MyThread[4];
        for (int i = 0; i < threadArray.length; i++) {
            threadArray[i] = new MyThread(myService);
        }
    }
}

```

```

        threadArray[i].start();
    }
}

```

程序运行结果如图 2-11 所示。

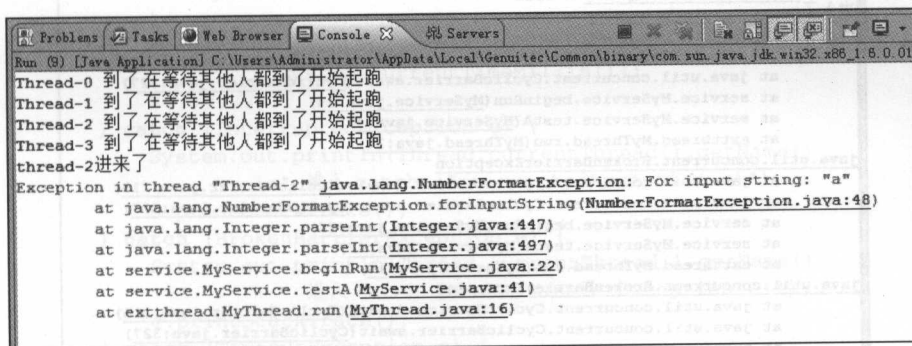


图 2-11 出现异常

从程序的运行结果来看，有一个线程出现异常报错，则其他线程继续等待，并不影响程序运行的主流程。

更改类 MyService.java 部分代码如下：

```

if (Thread.currentThread().getName().equals("Thread-2")) {
    System.out.println("thread-2 进来了");
    Thread.sleep(5000);
    Thread.currentThread().interrupt();
}

```

代码由原来出现异常改成了 interrupt() 中断。

程序运行结果如图 2-12 所示。

从运行结果来看，全部线程都进入了 catch 语句块，其中 Thread-2 线程进入了 InterruptedException 的 catch 语句块，其他 3 个线程进入了 BrokenBarrierException 的 catch 语句块。

类 CyclicBarrier 对于线程的中断 interrupt 处理会使用全有或者全无的破坏模型 (breakage model)，意思是如果有一个线程由于中断或者超时提前离开了屏障点，其他所有在屏障点等待的线程也会抛出 BrokenBarrierException 或者 InterruptedException 异常，并且离开屏障点。

2.2.5 方法 await(long timeout, TimeUnit unit) 超时出现异常的测试

方法 await(long timeout, TimeUnit unit) 的功能是如果在指定的时间内达到 parties 的数量，则程序继续向下运行，否则如果出现超时，则抛出 TimeoutException 异常。



图 2-12 异常信息

创建项目 CyclicBarrier_run3，创建类 MyService.java 代码如下：

```

package service;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class MyService {

    public CyclicBarrier cyclicBarrier = new CyclicBarrier(3, new Runnable() {
        @Override
        public void run() {
            System.out.println("彻底结束了 " + System.currentTimeMillis());
        }
    });

    public void testMethod() {
        try {
            System.out.println(Thread.currentThread().getName() + " 准备! "
                + System.currentTimeMillis());
            if (Thread.currentThread().getName().equals("Thread-0")) {

```

```

        System.out
            .println("Thread-0 执行了 cyclicBarrier.await(5, TimeUnit.
                SECONDS)");
        cyclicBarrier.await(5, TimeUnit.SECONDS);
    }
    if (Thread.currentThread().getName().equals("Thread-1")) {
        System.out.println("Thread-1 执行了 cyclicBarrier.await()");
        cyclicBarrier.await();
    }
    System.out.println(Thread.currentThread().getName() + " 开始! "
        + System.currentTimeMillis());
    ///////////
    } catch (InterruptedException e) {
        System.out.println(Thread.currentThread().getName()
            + " 进入 catch (InterruptedException e)");
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        System.out.println(Thread.currentThread().getName()
            + " 进入 catch (BrokenBarrierException e)");
        e.printStackTrace();
    } catch (TimeoutException e) {
        System.out.println(Thread.currentThread().getName()
            + " 进入 catch (TimeoutException e)");
        e.printStackTrace();
    }
}
}
}

```

线程类代码如图 2-13 所示。

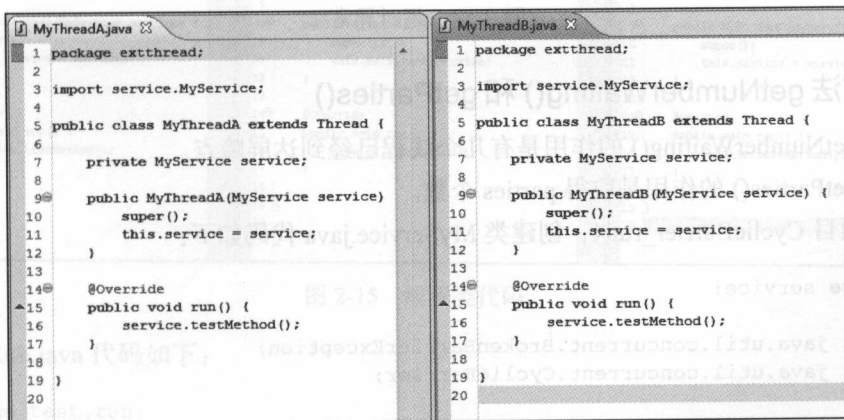


图 2-13 线程类代码

运行类 Run.java 代码如下：

```

package test.run;

import service.MyService;

```



```

import extthread.MyThreadA;
import extthread.MyThreadB;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        MyThreadA a = new MyThreadA(service);
        a.start();
        MyThreadB b = new MyThreadB(service);
        b.start();
    }
}

```

程序运行结果如图 2-14 所示。

```

<terminated> Run [Java Application] C:\accpMyEclipse8.5\Genuitec\Common\binary\com.sun.java.jdk.win32.x8
Thread-0 准备! 1418449245650
Thread-1 准备! 1418449245650
Thread-0 执行了 cyclicBarrier.await(5, TimeUnit.SECONDS)
Thread-1 执行了 cyclicBarrier.await()
Thread-0 进入 catch (TimeoutException e)
java.util.concurrent.TimeoutException
Thread-1 进入 catch (BrokenBarrierException e)
at java.util.concurrent.CyclicBarrier.dowait (CyclicBarrier.java:222)
at java.util.concurrent.CyclicBarrier.await (CyclicBarrier.java:399)
at service.MyService.testMethod (MyService.java:24)
at extthread.MyThreadA.run (MyThreadA.java:16)
java.util.concurrent.BrokenBarrierException
at java.util.concurrent.CyclicBarrier.dowait (CyclicBarrier.java:215)
at java.util.concurrent.CyclicBarrier.await (CyclicBarrier.java:327)
at service.MyService.testMethod (MyService.java:28)
at extthread.MyThreadB.run (MyThreadB.java:16)

```

图 2-14 超时异常

2.2.6 方法 getNumberWaiting() 和 getParties()

方法 getNumberWaiting() 的作用是有几个线程已经到达屏障点。

方法 getParties() 的作用是取得 parties 个数。

创建项目 CyclicBarrier_run4，创建类 MyService.java 代码如下：

```

package service;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class MyService {

    public CyclicBarrier cyclicBarrier = new CyclicBarrier(3, new Runnable() {
        @Override
        public void run() {
            System.out.println("                彻底结束了 ")
                + System.currentTimeMillis();
        }
    })
}

```

```

    });

    public void testMethod() {
        try {
            System.out.println(Thread.currentThread().getName() + " 准备! "
                + System.currentTimeMillis());
            if (Thread.currentThread().getName().equals("C")) {
                Thread.sleep(Integer.MAX_VALUE);
            }
            cyclicBarrier.await();
            System.out.println(Thread.currentThread().getName() + " 开始! "
                + System.currentTimeMillis());
            // //////////
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}

```

线程类代码如图 2-15 所示。

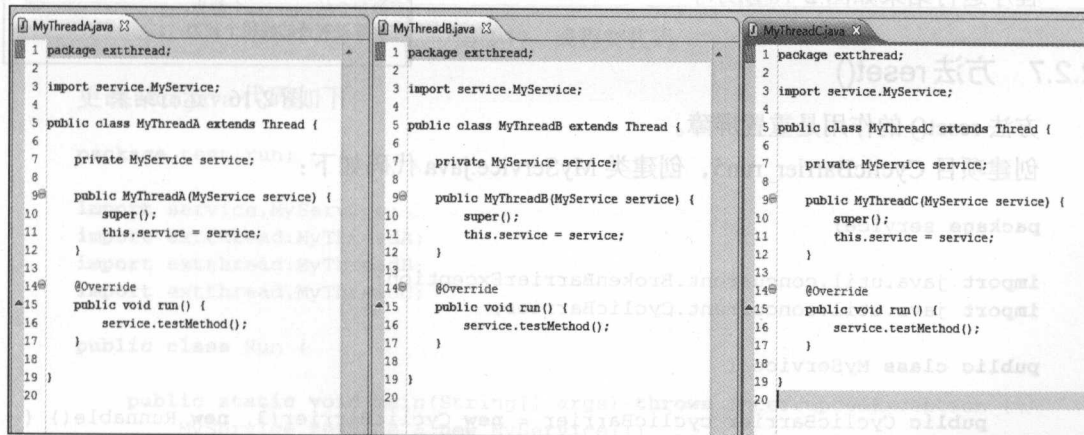


图 2-15 线程类代码

更改 Run.java 代码如下：

```

package test.run;

import service.MyService;
import extthread.MyThreadA;
import extthread.MyThreadB;
import extthread.MyThreadC;

public class Run {
    // ...
}

```

```

public static void main(String[] args) throws InterruptedException {
    MyService service = new MyService();
    MyThreadA a = new MyThreadA(service);
    a.setName("A");
    MyThreadB b = new MyThreadB(service);
    b.setName("B");
    MyThreadC c = new MyThreadC(service);
    c.setName("C");

    a.start();
    b.start();
    c.start();

    Thread.sleep(2000);
    System.out.println("屏障对象的parties个数为: "
        + service.cyclicBarrier.getParties());
    System.out.println("在屏障处等待的线程个数为: "
        + service.cyclicBarrier.getNumberWaiting());
}
}

```

程序运行结果如图 2-16 所示。

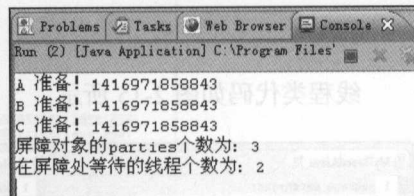


图 2-16 运行结果

2.2.7 方法 reset()

方法 reset() 的作用是重置屏障。

创建项目 CyclicBarrier_run5, 创建类 MyService.java 代码如下:

```

package service;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class MyService {

    public CyclicBarrier cyclicBarrier = new CyclicBarrier(3, new Runnable() {
        @Override
        public void run() {
            System.out.println("                彻底结束了 "
                + System.currentTimeMillis());
        }
    });

    public void testMethod() {
        try {
            System.out.println(Thread.currentThread().getName() + " 准备! "
                + System.currentTimeMillis());
            cyclicBarrier.await();
            System.out.println(Thread.currentThread().getName() + " 结束! "
                + System.currentTimeMillis());
        }
    }
}

```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        e.printStackTrace();
    }
}
}

```

线程类代码如图 2-17 所示。

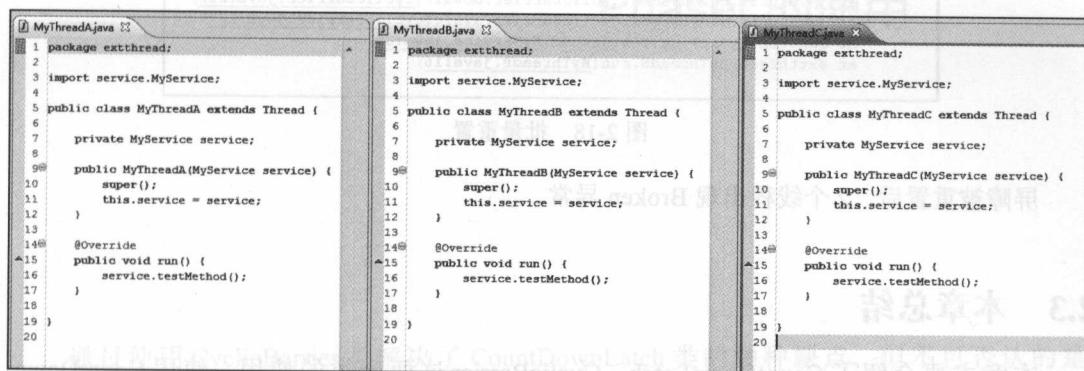


图 2-17 线程类代码

更改 Run.java 代码如下:

```

package test.run;

import service.MyService;
import extthread.MyThreadA;
import extthread.MyThreadB;
import extthread.MyThreadC;

public class Run {

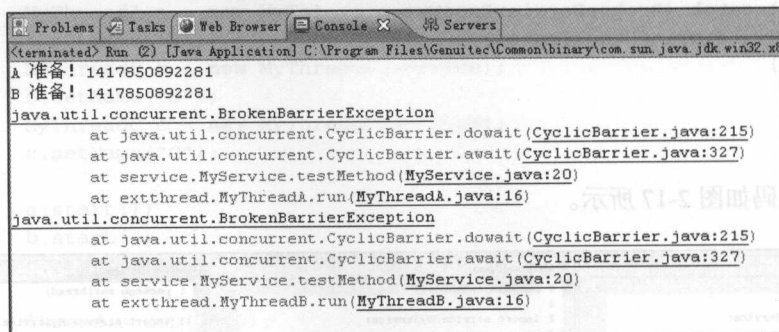
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        MyThreadA a = new MyThreadA(service);
        a.setName("A");
        MyThreadB b = new MyThreadB(service);
        b.setName("B");
        // 线程 C 未实例化
        a.start();
        b.start();

        Thread.sleep(2000);
        service.cyclicBarrier.reset();
    }
}

```

图 3-1 类 Phaser 的 API 列表

程序运行结果如图 2-18 所示。



```
<terminated> Run (2) [Java Application] C:\Program Files\Genuitec\Common\binary\com.sun.java.jdk.win32.x86
A 准备! 1417850892281
B 准备! 1417850892281
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:215)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)
    at service.MyService.testMethod(MyService.java:20)
    at extthread.MyThreadA.run(MyThreadA.java:16)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:215)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)
    at service.MyService.testMethod(MyService.java:20)
    at extthread.MyThreadB.run(MyThreadB.java:16)
```

图 2-18 批量重置

屏障被重置后，2 个线程出现 Broken 异常。

2.3 本章总结

本章主要介绍了 CountdownLatch、CyclicBarrier 这两个类的使用，使用 CountdownLatch 类可以实现两种角色的线程等待对方的效果，而 CyclicBarrier 类可以使同类线程互相等待达到同步的效果，使用这两个类可以更加完善地实现线程对象之间的同步性，对线程对象执行的轨迹控制更加方便。

Phaser 的使用

通过使用 `CyclicBarrier` 类解决了 `CountDownLatch` 类的种种缺点，但不可否认的是，`CyclicBarrier` 类还是有一些自身上的缺陷，比如不可以动态添加 parties 计数，调用一次 `await()` 方法仅仅占用 1 个 parties 计数，所以在 JDK1.7 中新增加了一个名称为 `Phaser` 的类来解决这样的问题。

类 `Phaser` 的全部 API 如图 3-1 所示。

```

● arrive() : int - Phaser
● arriveAndAwaitAdvance() : int - Phaser
● arriveAndDeregister() : int - Phaser
● awaitAdvance(int phase) : int - Phaser
● awaitAdvanceInterruptibly(int phase) : int - Phaser
● awaitAdvanceInterruptibly(int phase, long timeout, TimeUnit unit) : int - Phaser
● bulkRegister(int parties) : int - Phaser
● equals(Object obj) : boolean - Object
● forceTermination() : void - Phaser
● getArrivedParties() : int - Phaser
● getClass() : Class<?> - Object
● getParent() : Phaser - Phaser
● getPhase() : int - Phaser
● getRegisteredParties() : int - Phaser
● getRoot() : Phaser - Phaser
● getUnarrivedParties() : int - Phaser
● hashCode() : int - Object
● isTerminated() : boolean - Phaser
● notify() : void - Object
● notifyAll() : void - Object
● register() : int - Phaser
● toString() : String - Phaser
● wait() : void - Object
● wait(long timeout) : void - Object
● wait(long timeout, int nanos) : void - Object

```

图 3-1 类 `Phaser` 的 API 列表

在后面的章节将介绍 Phaser 类中主要的方法，熟练掌握 Phaser 类的使用是熟练掌握 JDK 并发包的必要知识点。

3.1 Phaser 的使用

单词 Phaser 的发音为 ['feɪzə]，中文翻译为移相器。移相器这个术语是在电子专业中使用的，但在 Java 语言中，该类是在 JDK1.7 版本中新增的，所以如果想使用它，必须安装 JDK 是 1.7 的版本。

类 Phaser 拥有的方法列表如图 3-2 所示。

类 Phaser 对计数的操作是加法操作。

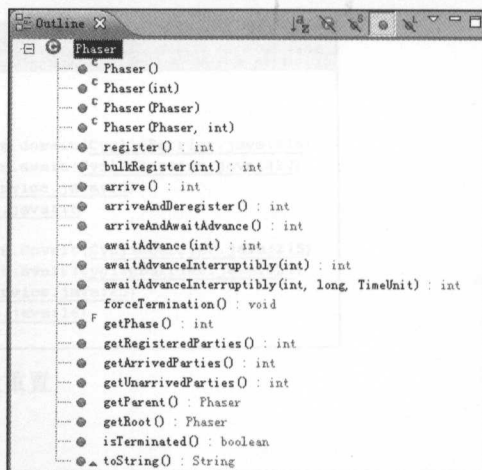


图 3-2 Eclipse 中的大纲视图方法列表

3.2 类 Phaser 的 arriveAndAwaitAdvance() 方法测试 1

方法 arriveAndAwaitAdvance() 的作用与 CountdownLatch 类中的 await() 方法大体一样，通过从方法的名称解释来看，arrive 是到达的意思，wait 是等待的意思，而 advance 是前进、促进的意思，所以执行这个方法的作用就是当前线程已经到达屏障，在此等待一段时间，等条件满足后继续向下一个屏障继续执行。

通过前面的解释可以发现，类 Phaser 具有设置多屏障的功能，有些类似于体育竞赛中“赛段”的作用，运动员第一赛段结束后，开始休整准备，然后集体到达第二赛段的起跑点，等待比赛开始后，运动员们又继续比赛了，说明 Phaser 类与 CyclicBarrier 类在功能上有重叠。下面的章节将使用 Phaser 来实现一个比赛过程中的“多赛段”问题。

创建测试用的项目 Phaser_test1，创建 PrintTools.java 类代码如下：

```
package tools;

import java.util.concurrent.Phaser;

public class PrintTools {

    public static Phaser phaser;

    public static void methodA() {
        System.out.println(Thread.currentThread().getName() + " A1 begin="
            + System.currentTimeMillis());
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " A1 end="
            + System.currentTimeMillis());

        System.out.println(Thread.currentThread().getName() + " A2 begin="
```

```

        + System.currentTimeMillis());
    phaser.arriveAndAwaitAdvance();
    System.out.println(Thread.currentThread().getName() + " A2   end="
        + System.currentTimeMillis());
}

public static void methodB() {
    try {
        System.out.println(Thread.currentThread().getName() + " A1 begin="
            + System.currentTimeMillis());
        Thread.sleep(5000);
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " A1   end="
            + System.currentTimeMillis());

        System.out.println(Thread.currentThread().getName() + " A2 begin="
            + System.currentTimeMillis());
        Thread.sleep(5000);
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " A2   end="
            + System.currentTimeMillis());
    } catch (InterruptedException e) {
        //TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}

```

线程类代码如图 3-3 所示。

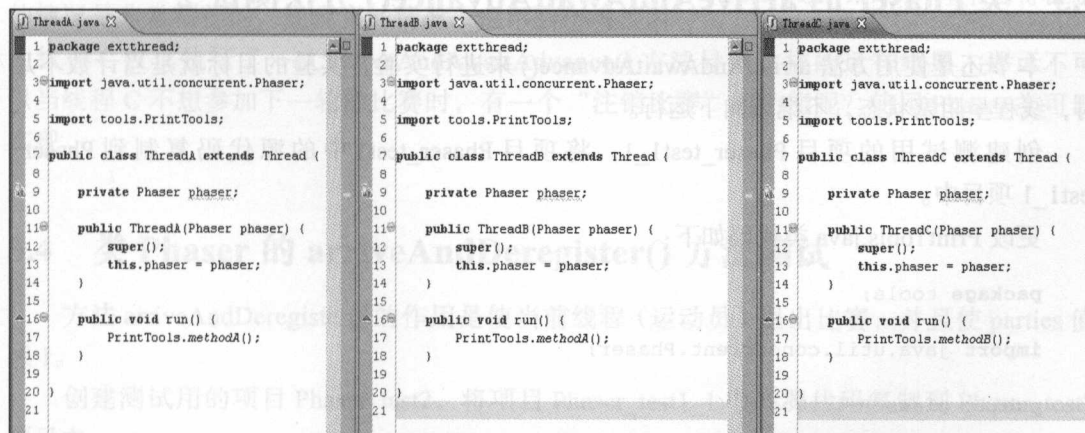


图 3-3 线程类代码

运行类 Run.java 代码如下:

```
package test;
```

```

import java.util.concurrent.Phaser;

import tools.PrintTools;
import extthread.ThreadA;
import extthread.ThreadB;
import extthread.ThreadC;

public class Run {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(3);
        PrintTools.phaser = phaser;

        ThreadA a = new ThreadA(phaser);
        a.setName("A");
        a.start();

        ThreadB b = new ThreadB(phaser);
        b.setName("B");
        b.start();

        ThreadC c = new ThreadC(phaser);
        c.setName("C");
        c.start();
    }
}

```

程序运行后的效果如图 3-4 所示。

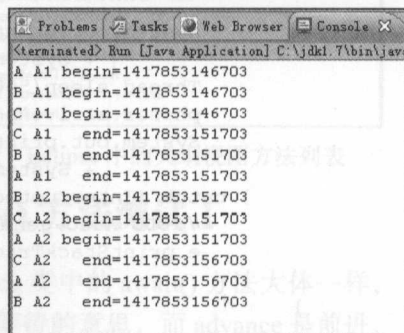


图 3-4 运行结果

3.3 类 Phaser 的 arriveAndAwaitAdvance() 方法测试 2

本节还是使用方法 arriveAndAwaitAdvance() 来进行实验，实验的目标就是当计数不足时，线程呈阻塞状态，不继续向下运行。

创建测试用的项目 Phaser_test1_1，将项目 Phaser_test1 中的源代码复制到 Phaser_test1_1 项目中。

更改 PrintTools.java 类代码如下：

```

package tools;

import java.util.concurrent.Phaser;

public class PrintTools {

    public static Phaser phaser;

    public static void methodA() {
        System.out.println(Thread.currentThread().getName() + " A1 begin="
            + System.currentTimeMillis());
        phaser.arriveAndAwaitAdvance();
    }
}

```

```

        System.out.println(Thread.currentThread().getName() + " A1 end="
            + System.currentTimeMillis());

        System.out.println(Thread.currentThread().getName() + " A2 begin="
            + System.currentTimeMillis());
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " A2 end="
            + System.currentTimeMillis());
    }

    public static void methodB() {
        try {
            System.out.println(Thread.currentThread().getName() + " A1 begin="
                + System.currentTimeMillis());
            Thread.sleep(5000);
            phaser.arriveAndAwaitAdvance();
            System.out.println(Thread.currentThread().getName() + " A1 end="
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            //TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

程序运行后的效果如图 3-5 所示。

从运行结果来看，说明线程 C，也就是运动员 C，中途退出了比赛，导致后面的比赛不能正常继续，这样的情况是非常糟糕的，在现实的情况下也不会出现，因为线程 C 仅仅执行了一次 `arriveAndAwaitAdvance()` 方法导致这样的运行结果，那可不可以当线程 C 不想参加下一轮的比赛时，有一个“注销比赛”的功能呢？使用 `Phaser` 类可以实现。

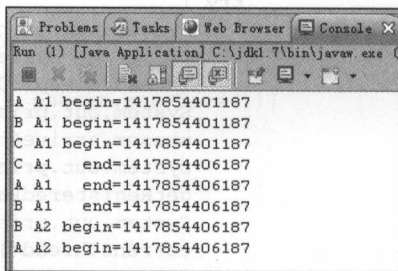


图 3-5 运行结果是一直持等待状态

3.4 类 `Phaser` 的 `arriveAndDeregister()` 方法测试

方法 `arriveAndDeregister()` 的作用是使当前线程（运动员）退出比赛，并且使 `parties` 值减 1。

创建测试用的项目 `Phaser_test2`，将项目 `Phaser_test1_1` 中的源代码复制到 `Phaser_test2` 项目中。

更改 `PrintTools.java` 类代码如下：

```

package tools;

import java.util.concurrent.Phaser;

```

```

public class PrintTools {

    public static Phaser phaser;

    public static void methodA() {
        System.out.println(Thread.currentThread().getName() + " A1 begin="
            + System.currentTimeMillis());
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " A1  end="
            + System.currentTimeMillis());

        System.out.println(Thread.currentThread().getName() + " A2 begin="
            + System.currentTimeMillis());
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " A2  end="
            + System.currentTimeMillis());
    }

    public static void methodB() {
        try {
            System.out.println(Thread.currentThread().getName() + " A1 begin="
                + System.currentTimeMillis());
            Thread.sleep(5000);
            System.out.println("A: " + phaser.getRegisteredParties());
            phaser.arriveAndDeregister();
            System.out.println("B: " + phaser.
                getRegisteredParties());
            System.out.println(Thread.
                currentThread().getName() + " A1  end="
                    + System.currentTimeMillis());
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

程序运行后的效果如图 3-6 所示。

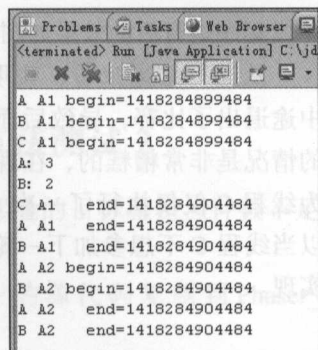


图 3-6 运行结果

3.5 类 Phaser 的 getPhase() 和 onAdvance() 方法测试

方法 getPhase() 获取的是已经到达第几个屏障。

创建测试用的项目 Phaser_test3, 创建 ThreadA.java 类代码如下:

```

package extthread;

import java.util.concurrent.Phaser;

public class ThreadA extends Thread {

```

```

private Phaser phaser;

public ThreadA(Phaser phaser) {
    super();
    this.phaser = phaser;
}

public void run() {
    System.out.println("A begin");
    phaser.arriveAndAwaitAdvance();
    System.out.println("A end phase value=" + phaser.getPhase());

    System.out.println("A begin");
    phaser.arriveAndAwaitAdvance();
    System.out.println("A end phase value=" + phaser.getPhase());

    System.out.println("A begin");
    phaser.arriveAndAwaitAdvance();
    System.out.println("A end phase value=" + phaser.getPhase());

    System.out.println("A begin");
    phaser.arriveAndAwaitAdvance();
    System.out.println("A end phase value=" + phaser.getPhase());
}
}

```

运行类 Run.java 代码如下:

```

package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;

public class Run {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(1);
        ThreadA a = new ThreadA(phaser);
        a.start();
    }
}

```

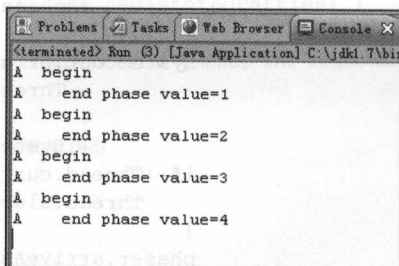


图 3-7 运行结果

程序运行后的效果如图 3-7 所示。

方法 onAdvance() 的作用是通过新的屏障时被调用。

创建测试用的项目 Phaser_onAdvance, 类代码如下:

```

package service;

import java.util.concurrent.Phaser;

```



```

public class MyService {
    private Phaser phaser;

    public MyService(Phaser phaser) {
        super();
        this.phaser = phaser;
    }

    public void testMethod() {
        try {
            System.out.println("A begin ThreadName="
                + Thread.currentThread().getName()
                + " "
                + System.currentTimeMillis());
            if (Thread.currentThread().getName().equals("B")) {
                Thread.sleep(5000);
            }
            phaser.arriveAndAwaitAdvance();
            System.out.println("A end ThreadName="
                + Thread.currentThread().getName() + " end phase value="
                + phaser.getPhase() + " " + System.currentTimeMillis());
            //////////
            System.out.println("B begin ThreadName="
                + Thread.currentThread().getName()
                + " "
                + System.currentTimeMillis());
            if (Thread.currentThread().getName().equals("B")) {
                Thread.sleep(5000);
            }
            phaser.arriveAndAwaitAdvance();
            System.out.println("B end ThreadName="
                + Thread.currentThread().getName() + " end phase value="
                + phaser.getPhase() + " " + System.currentTimeMillis());
            //////////
            System.out.println("C begin ThreadName="
                + Thread.currentThread().getName()
                + " "
                + System.currentTimeMillis());
            if (Thread.currentThread().getName().equals("B")) {
                Thread.sleep(5000);
            }
            phaser.arriveAndAwaitAdvance();
            System.out.println("C end ThreadName="
                + Thread.currentThread().getName() + " end phase value="
                + phaser.getPhase() + " " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

线程类代码如图 3-8 所示。

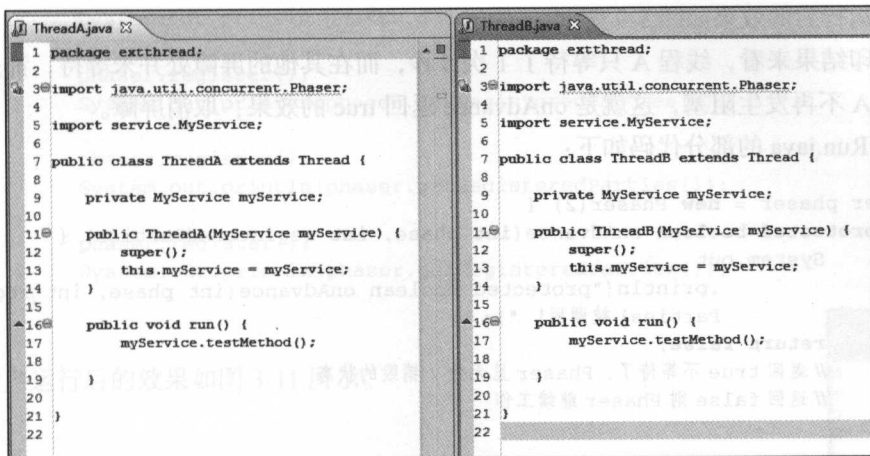


图 3-8 线程类代码

运行类 Run.java 代码如下:

```

package test;

import java.util.concurrent.Phaser;

import service.MyService;
import extthread.ThreadA;
import extthread.ThreadB;

public class Run {

    public static void main(String[] args) {
        Phaser phaser = new Phaser(2) {
            protected boolean onAdvance(int phase, int registeredParties) {
                System.out
                    .println("protected boolean onAdvance(int phase, int registered
                        Parties) 被调用!");
                return true;
                // 返回 true 不等待了, Phaser 呈无效 / 销毁的状态
                // 返回 false 则 Phaser 继续工作
            }
        };

        MyService service = new MyService(phaser);

        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();
    }
}

```

程序运行后的效果如图 3-9 所示。

从打印结果来看，线程 A 只等待了 1 次 5 秒，而在其他的屏障处并未等待，都是快速打印，线程 A 不再发生阻塞，这就是 onAdvance 返回 true 的效果，取消屏障。

更改 Run.java 的部分代码如下：

```
Phaser phaser = new Phaser(2) {
    protected boolean onAdvance(int phase, int registeredParties) {
        System.out
            .println("protected boolean onAdvance(int phase, int registered
                Parties) 被调用！");
        return false;
        // 返回 true 不等待了，Phaser 呈无效 / 销毁的状态
        // 返回 false 则 Phaser 继续工作
    }
};
```

返回值改成 return false；

程序运行结果如图 3-10 所示。

```
A begin ThreadName=A          1418454089295
A begin ThreadName=B          1418454089295
protected boolean onAdvance(int phase, int registeredParties) 被调用!
A end ThreadName=B end phase value=-2147483647 1418454094295
B begin ThreadName=B          1418454094295
A end ThreadName=A end phase value=-2147483647 1418454094295
B begin ThreadName=A          1418454094295
B end ThreadName=A end phase value=-2147483647 1418454094295
C begin ThreadName=A          1418454094295
C end ThreadName=A end phase value=-2147483647 1418454094295
B end ThreadName=B end phase value=-2147483647 1418454099295
C begin ThreadName=B          1418454099295
C end ThreadName=B end phase value=-2147483647 1418454104296
```

图 3-9 屏障功能被取消

```
A begin ThreadName=A          14184544416721
A begin ThreadName=B          14184544416721
protected boolean onAdvance(int phase, int registeredParties) 被调用!
A end ThreadName=B end phase value=1 1418454421722
A end ThreadName=A end phase value=1 1418454421722
B begin ThreadName=B          14184544421722
B begin ThreadName=A          1418454421722
protected boolean onAdvance(int phase, int registeredParties) 被调用!
B end ThreadName=B end phase value=2 1418454426722
B end ThreadName=A end phase value=2 1418454426722
C begin ThreadName=B          14184544426722
C begin ThreadName=A          1418454426722
protected boolean onAdvance(int phase, int registeredParties) 被调用!
C end ThreadName=B end phase value=3 1418454431722
C end ThreadName=A end phase value=3 1418454431722
```

图 3-10 屏障功能继续使用

3.6 类 Phaser 的 getRegisteredParties() 方法和 register() 测试

方法 getRegisteredParties() 获得注册的 parties 数量。

每执行一次方法 register() 就动态添加一个 parties 值。

创建测试用的项目 Phaser_test4，创建 Run.java 类代码如下：

```
package test;

import java.util.concurrent.Phaser;

public class Run {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(5);
        System.out.println(phaser.getRegisteredParties());

        phaser.register();
```

```

System.out.println(phaser.getRegisteredParties());

phaser.register();
System.out.println(phaser.getRegisteredParties());

phaser.register();
System.out.println(phaser.getRegisteredParties());

phaser.register();
System.out.println(phaser.getRegisteredParties());
}

```

程序运行后的效果如图 3-11 所示。

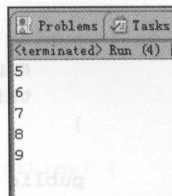


图 3-11 运行结果

3.7 类 Phaser 的 bulkRegister() 方法测试

方法 bulkRegister() 可以批量增加 parties 数量。

创建测试用的项目 Phaser_test4_1, 创建 Run.java 类代码如下:

```

package test;

import java.util.concurrent.Phaser;

public class Run {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(10);
        System.out.println(phaser.getRegisteredParties());

        phaser.bulkRegister(10);
        System.out.println(phaser.getRegisteredParties());

        phaser.bulkRegister(10);
        System.out.println(phaser.getRegisteredParties());

        phaser.bulkRegister(10);
        System.out.println(phaser.getRegisteredParties());

        phaser.bulkRegister(10);
        System.out.println(phaser.getRegisteredParties());
    }
}

```

程序运行后的效果如图 3-12 所示。

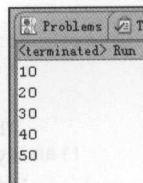


图 3-12 运行结果

3.8 类 Phaser 的 getArrivedParties() 和 getUnarrivedParties() 方法测试

方法 getArrivedParties() 获得已经被使用的 parties 个数。

方法 `getUnarrivedParties()` 获得未被使用的 `parties` 个数。

创建测试用的项目 `Phaser_test5`，线程类 `MyThread.java` 代码如下：

```
package extthread;

import java.util.concurrent.Phaser;

public class MyThread extends Thread {

    private Phaser phaser;

    public MyThread(Phaser phaser) {
        super();
        this.phaser = phaser;
    }

    public void run() {
        System.out.println(Thread.currentThread().getName() + " A1 begin="
            + System.currentTimeMillis());
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " A1 end="
            + System.currentTimeMillis());
    }
}
```

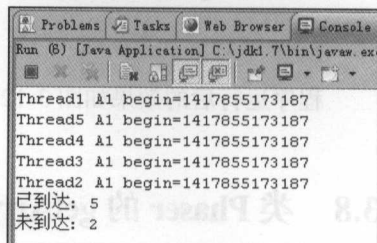
创建 `Run.java` 类代码如下：

```
package test;

import java.util.concurrent.Phaser;

import extthread.MyThread;

public class Run {
    public static void main(String[] args) throws InterruptedException {
        Phaser phaser = new Phaser(7);
        MyThread[] myThreadArray = new MyThread[5];
        for (int i = 0; i < myThreadArray.length; i++) {
            myThreadArray[i] = new MyThread(phaser);
            myThreadArray[i].setName("Thread" + (i + 1));
            myThreadArray[i].start();
        }
        Thread.sleep(2000);
        System.out.println("已到达：" + phaser.
            getArrivedParties());
        System.out.println("未到达：" + phaser.
            getUnarrivedParties());
    }
}
```



程序运行后的效果如图 3-13 所示。

图 3-13 运行结果

3.9 类 Phaser 的 arrive() 方法测试 1

方法 arrive() 的作用是使 parties 值加 1, 并且不在屏障处等待, 直接向下面的代码继续运行, 并且 Phaser 类有计数重置功能。

创建测试用的项目 Phaser_test6_1, 创建 Run.java 类代码如下:

```
package test;

import java.util.concurrent.Phaser;

public class Run {

    public static void main(String[] args) {
        Phaser phaser = new Phaser(2) {
            protected boolean onAdvance(int phase, int registeredParties) {
                System.out.println(" 到达了未通过! phase=" + phase
                    + " registeredParties=" + registeredParties);
                return super.onAdvance(phase, registeredParties);
            };
        };

        System.out.println("A1 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());
        phaser.arrive();
        System.out.println("A1 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());

        System.out.println("A2 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());
        phaser.arrive();
        System.out.println("A2 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());
        //////////////////////////////////

        System.out.println("B1 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());
        phaser.arrive();
        System.out.println("B1 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());

        System.out.println("B2 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());
        phaser.arrive();
        System.out.println("B2 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties());
    }
}
```



```

        + " getArrivedParties=" + phaser.getArrivedParties());
        ///////////////
        System.out.println("C1 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());
        phaser.arrive();
        System.out.println("C1 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());
        System.out.println("C2 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());
        phaser.arrive();
        System.out.println("C2 getPhase=" + phaser.getPhase()
            + " getRegisteredParties=" + phaser.getRegisteredParties()
            + " getArrivedParties=" + phaser.getArrivedParties());
        ///////////////
    }
}

```

程序运行后的效果如图 3-14 所示。

方法 `arrive()` 的功能是使 `getArrivedParties()` 计数加 1，不等待其他线程到达屏障。

在控制台中多次出现 `getArrivedParties=0` 的运行结果，所以可以分析出 `Phaser` 类在经过屏障点后计数能被重置。

```

A1 getPhase=0 getRegisteredParties=2 getArrivedParties=0
A1 getPhase=0 getRegisteredParties=2 getArrivedParties=1
A2 getPhase=0 getRegisteredParties=2 getArrivedParties=1
到达了未通过! phase=0 registeredParties=2
A2 getPhase=1 getRegisteredParties=2 getArrivedParties=0
B1 getPhase=1 getRegisteredParties=2 getArrivedParties=0
B1 getPhase=1 getRegisteredParties=2 getArrivedParties=1
B2 getPhase=1 getRegisteredParties=2 getArrivedParties=1
到达了未通过! phase=1 registeredParties=2
B2 getPhase=2 getRegisteredParties=2 getArrivedParties=0
C1 getPhase=2 getRegisteredParties=2 getArrivedParties=0
C1 getPhase=2 getRegisteredParties=2 getArrivedParties=1
C2 getPhase=2 getRegisteredParties=2 getArrivedParties=1
到达了未通过! phase=2 registeredParties=2
C2 getPhase=3 getRegisteredParties=2 getArrivedParties=0

```

图 3-14 运行结果

3.10 类 `Phaser` 的 `arrive()` 方法测试 2

本节的实验目标还是测试当计数不足时，线程 A 和 B 依然呈等待状态。

创建测试用的项目 `Phaser_test6_2`，创建类 `MyService.java` 代码如下：

```

package service;

import java.util.concurrent.Phaser;

public class MyService {

    public Phaser phaser;

    public MyService(Phaser phaser) {
        super();
        this.phaser = phaser;
    }
}

```

```

public void testMethodA() {
    try {
        System.out.println(Thread.currentThread().getName() + " begin A1 "
            + System.currentTimeMillis());
        Thread.sleep(3000);
        System.out.println(phaser.getArrivedParties());
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " end A1 "
            + System.currentTimeMillis());

        System.out.println(Thread.currentThread().getName() + " begin A2 "
            + System.currentTimeMillis());
        Thread.sleep(3000);
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " end A2 "
            + System.currentTimeMillis());

        System.out.println(Thread.currentThread().getName() + " begin A3 "
            + System.currentTimeMillis());
        Thread.sleep(3000);
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " end A3 "
            + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public void testMethodB() {
    System.out.println(Thread.currentThread().getName() + " begin A1 "
        + System.currentTimeMillis());
    phaser.arrive();
    System.out.println(Thread.currentThread().getName() + " end A1 "
        + System.currentTimeMillis());

    System.out.println(Thread.currentThread().getName() + " begin A2 "
        + System.currentTimeMillis());
    phaser.arrive();
    System.out.println(Thread.currentThread().getName() + " end A2 "
        + System.currentTimeMillis());

    System.out.println(Thread.currentThread().getName() + " begin A3 "
        + System.currentTimeMillis());
    phaser.arrive();
    System.out.println(Thread.currentThread().getName() + " end A3 "
        + System.currentTimeMillis());
}
}

```

线程类代码如图 3-15 所示。

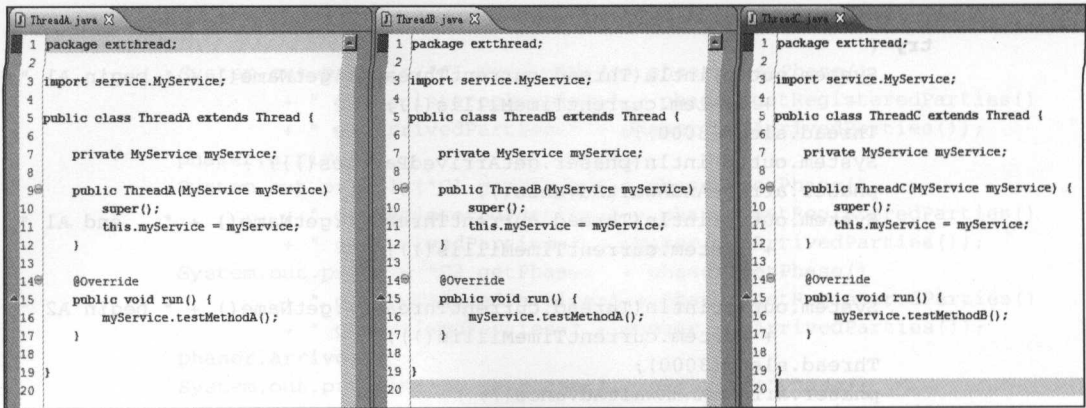


图 3-15 线程类代码

运行类 Run.java 代码如下：

```
package test;

import java.util.concurrent.Phaser;

import service.MyService;
import extthread.ThreadA;
import extthread.ThreadB;
import extthread.ThreadC;

public class Run {

    public static void main(String[] args) {
        Phaser phaser = new Phaser(3);
        MyService service = new MyService(phaser);

        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();

        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();

        ThreadC c = new ThreadC(service);
        c.setName("C");
        c.start();
    }
}
```

程序运行后的效果如图 3-16 所示。

线程 C 在 parties 计数达到 3 后自动重置成 0，线程

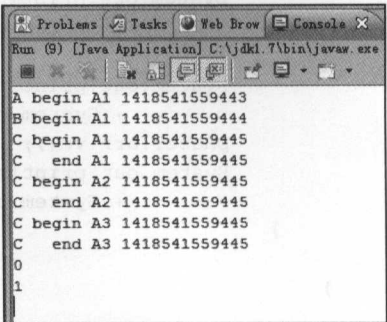


图 3-16 运行结果

A 和 B 由于达不到 parties 为 3 的情况，所以它们俩一直在等待。

3.11 类 Phaser 的 awaitAdvance(int phase) 方法测试

方法 awaitAdvance(int Phase) 的作用是：如果传入参数 phase 值和当前 getPhase() 方法返回值一样，则在屏障处等待，否则继续向下面运行，有些类似于旁观者的作用，当观察的条件满足了就等待（旁观），如果条件不满足，则程序向下继续运行。

创建测试用的项目 Phaser_test7，创建 2 个线程类代码如图 3-17 所示。

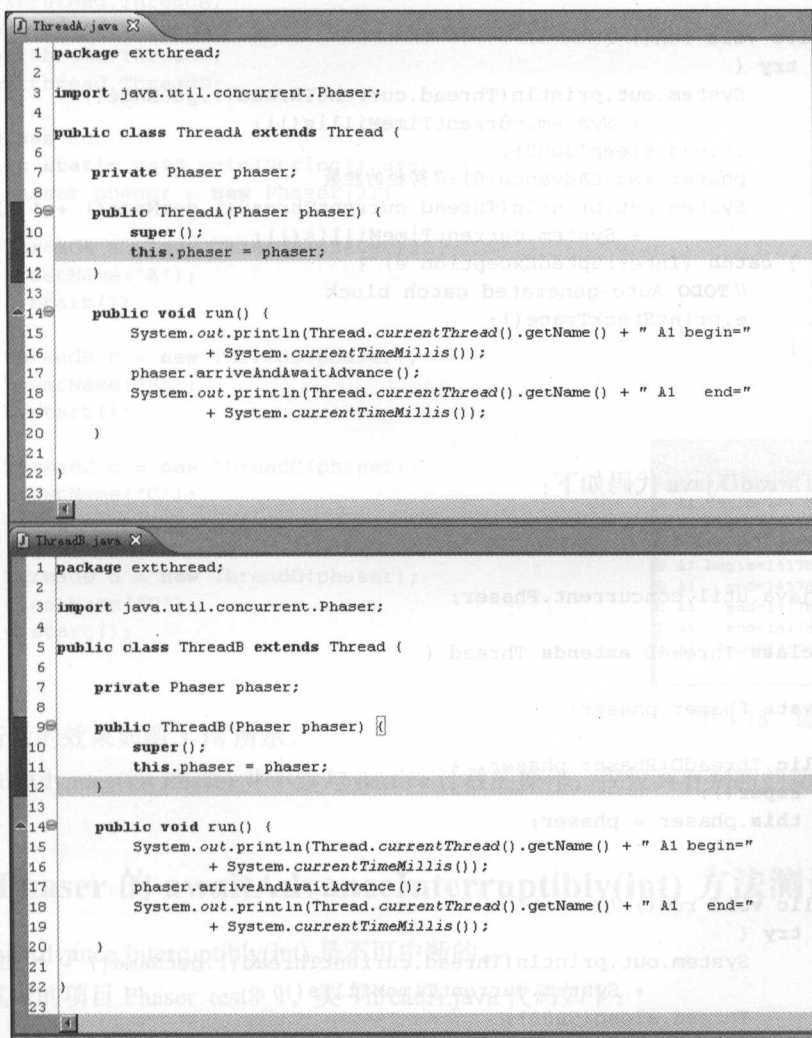


图 3-17 线程 AB 代码

线程类 ThreadC.java 代码如下：

```

package extthread;

import java.util.concurrent.Phaser;

public class ThreadC extends Thread {

    private Phaser phaser;

    public ThreadC(Phaser phaser) {
        super();
        this.phaser = phaser;
    }

    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " A1 begin="
                + System.currentTimeMillis());
            Thread.sleep(3000);
            phaser.awaitAdvance(0); // 跨栏的栏数
            System.out.println(Thread.currentThread().getName() + " A1 end="
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

线程类 ThreadD.java 代码如下：

```

package extthread;

import java.util.concurrent.Phaser;

public class ThreadD extends Thread {

    private Phaser phaser;

    public ThreadD(Phaser phaser) {
        super();
        this.phaser = phaser;
    }

    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " A1 begin="
                + System.currentTimeMillis());
            Thread.sleep(5000);
            phaser.arriveAndAwaitAdvance();
            System.out.println(Thread.currentThread().getName() + " A1 end="
                + System.currentTimeMillis());
        } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }
}

```

运行类 Run.java 代码如下:

```

package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;
import extthread.ThreadB;
import extthread.ThreadC;
import extthread.ThreadD;

public class Run {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(3);

        ThreadA a = new ThreadA(phaser);
        a.setName("A");
        a.start();

        ThreadB b = new ThreadB(phaser);
        b.setName("B");
        b.start();

        ThreadC c = new ThreadC(phaser);
        c.setName("C");
        c.start();

        ThreadD d = new ThreadD(phaser);
        d.setName("D");
        d.start();
    }
}

```

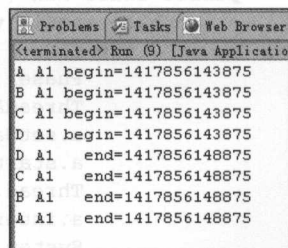


图 3-18 运行结果

程序运行后的效果如图 3-18 所示。

方法 `awaitAdvance(int Phase)` 并不参与 parties 计数的操作, 仅仅具有判断的功能。

3.12 类 Phaser 的 `awaitAdvanceInterruptibly(int)` 方法测试 1

方法 `awaitAdvance Interruptibly(int)` 是不可中断的。

创建测试用的项目 `Phaser_test8_1`, 类 `ThreadA.java` 代码如下:

```

package extthread;

import java.util.concurrent.Phaser;

```



```

public class ThreadA extends Thread {

    private Phaser phaser;

    public ThreadA(Phaser phaser) {
        super();
        this.phaser = phaser;
    }

    public void run() {
        System.out.println(Thread.currentThread().getName() + " A1 begin="
            + System.currentTimeMillis());
        phaser.awaitAdvance(0);
        System.out.println(Thread.currentThread().getName() + " A1 end="
            + System.currentTimeMillis());
    }
}

```

运行类 Run.java 代码如下：

```

package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;

public class Run {
    public static void main(String[] args) {
        try {
            Phaser phaser = new Phaser(3);
            ThreadA a = new ThreadA(phaser);
            a.setName("A");
            a.start();
            Thread.sleep(5000);
            a.interrupt();
            System.out.println("中断了c");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 3-19 所示。

控制台并没有出现异常，说明线程并未中断。

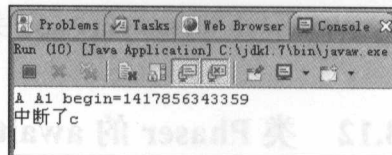


图 3-19 运行结果

3.13 类 Phaser 的 awaitAdvanceInterruptibly(int) 方法测试 2

方法 awaitAdvanceInterruptibly(int) 是可中断的。

创建测试用的项目 Phaser_test8_2，类 ThreadA.java 代码如下：

```

package extthread;

import java.util.concurrent.Phaser;

public class ThreadA extends Thread {

    private Phaser phaser;

    public ThreadA(Phaser phaser) {
        super();
        this.phaser = phaser;
    }

    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " A1 begin="
                + System.currentTimeMillis());
            phaser.awaitAdvanceInterruptibly(0); // 符合栏数就 wait
            System.out.println(Thread.currentThread().getName() + " A1 end="
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            System.out.println(" 进入 catch");
            e.printStackTrace();
        }
    }
}

```

运行类 Run.java 代码如下:

```

package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;

public class Run {

    public static void main(String[] args) {
        try {
            Phaser phaser = new Phaser(3);
            ThreadA a = new ThreadA(phaser);
            a.setName("A");
            a.start();
            Thread.sleep(5000);
            a.interrupt();
            System.out.println(" 中断了 c");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 3-20 所示。

```

A1 begin=1417856449765
中断了c
java.lang.InterruptedException
进入catch
    at java.util.concurrent.Phaser.awaitAdvanceInterruptibly(Phaser.java:750)
    at extthread.ThreadA.run(ThreadA.java:18)

```

图 3-20 运行结果

控制台出现异常，线程被中断了。

3.14 类 Phaser 的 awaitAdvanceInterruptibly(int) 方法测试 3

方法 awaitAdvanceInterruptibly(int) 的作用是当线程执行的栏数不符合指定的参数值时，则继续执行下面的代码。

创建测试用的项目 Phaser_test8_3，类 ThreadA.java 代码如下：

```

package extthread;

import java.util.concurrent.Phaser;

public class ThreadA extends Thread {

    private Phaser phaser;

    public ThreadA(Phaser phaser) {
        super();
        this.phaser = phaser;
    }

    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " A1 begin="
                + System.currentTimeMillis());
            phaser.awaitAdvanceInterruptibly(10); // 不符合栏数就继续运行
            System.out.println(Thread.currentThread().getName() + " A1 end="
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            System.out.println(" 进入 catch");
            e.printStackTrace();
        }
    }
}

```

运行类 Run.java 代码如下：

```

package test;

import java.util.concurrent.Phaser;

```

```
import extthread.ThreadA;

public class Run {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(3);
        ThreadA a = new ThreadA(phaser);
        a.setName("A");
        a.start();
    }
}
```

程序运行结果如图 3-21 所示。

程序快速继续向下运行，运行的时间都是一样的，继续向下运行的原因是栏数不符合 10 个。

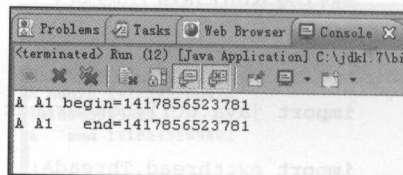


图 3-21 运行结果

3.15 类 Phaser 的 awaitAdvanceInterruptibly(int,long,TimeUnit) 方法测试 4

方法 awaitAdvanceInterruptibly(int,long,TimeUnit) 的作用是在指定的栏数等待最大的单位时间，如果在指定的时间内，栏数未变，则出现异常，否则继续向下运行。

创建测试用的项目 Phaser_test8_4，类 ThreadA.java 代码如下：

```
package extthread;

import java.util.concurrent.Phaser;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class ThreadA extends Thread {
    private Phaser phaser;

    public ThreadA(Phaser phaser) {
        super();
        this.phaser = phaser;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " begin "
                + System.currentTimeMillis());
            phaser.awaitAdvanceInterruptibly(0, 5, TimeUnit.SECONDS);
            System.out.println(Thread.currentThread().getName() + " end "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.out.println("InterruptedException e");
        }
    }
}
```

```

    } catch (TimeoutException e) {
        e.printStackTrace();
        System.out.println("TimeoutException e");
    }
}
}

```

运行类 Run1.java 代码如下：

```

package test;

import java.util.concurrent.Phaser;

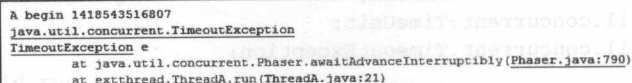
import extthread.ThreadA;

public class Run1 {

    public static void main(String[] args) {
        Phaser phaser = new Phaser(3);
        ThreadA a = new ThreadA(phaser);
        a.setName("A");
        a.start();
    }
}

```

程序运行结果如图 3-22 所示。



```

A begin 1418543516807
java.util.concurrent.TimeoutException
TimeoutException e
    at java.util.concurrent.Phaser.awaitAdvanceInterruptibly(Phaser.java:790)
    at extthread.ThreadA.run(ThreadA.java:21)

```

图 3-22 超时了出现异常

因为 5 秒之后 phaser 阶段值并没有发生改变。

运行类 Run2.java 代码如下：

```

package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;

public class Run2 {

    public static void main(String[] args) throws InterruptedException {
        Phaser phaser = new Phaser(3);
        ThreadA a = new ThreadA(phaser);
        a.setName("A");
        a.start();
    }
}

```

```

Thread.sleep(1000);
phaser.arrive();
Thread.sleep(1000);
phaser.arrive();
Thread.sleep(1000);
phaser.arrive();
System.out.println(System.currentTimeMillis());
}
}

```

程序运行结果如图 3-23 所示。

运行类 Run3.java 代码如下：

```

package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;

public class Run3 {

    public static void main(String[] args) throws InterruptedException {
        Phaser phaser = new Phaser(3);
        ThreadA a = new ThreadA(phaser);
        a.setName("A");
        a.start();
        Thread.sleep(1000);
        a.interrupt();
    }
}

```

程序运行结果如图 3-24 所示。

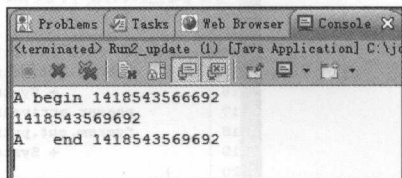


图 3-23 未超时

```

A begin 1418543638673
java.lang.InterruptedException
InterruptedException e
    at java.util.concurrent.Phaser.awaitAdvanceInterruptibly(Phaser.java:788)
    at extthread.ThreadA.run(ThreadA.java:21)

```

图 3-24 停止了

出现异常的原因是提前将还未到达 5 秒的线程进行了中断。

3.16 类 Phaser 的 forceTermination() 和 isTerminated() 方法测试

方法 forceTermination() 使 Phaser 对象的屏障功能失效，而方法 isTerminated() 是判断 Phaser 对象是否已经呈销毁状态。

创建测试用的项目 Phaser_test9，创建两个线程类代码如图 3-25 所示。

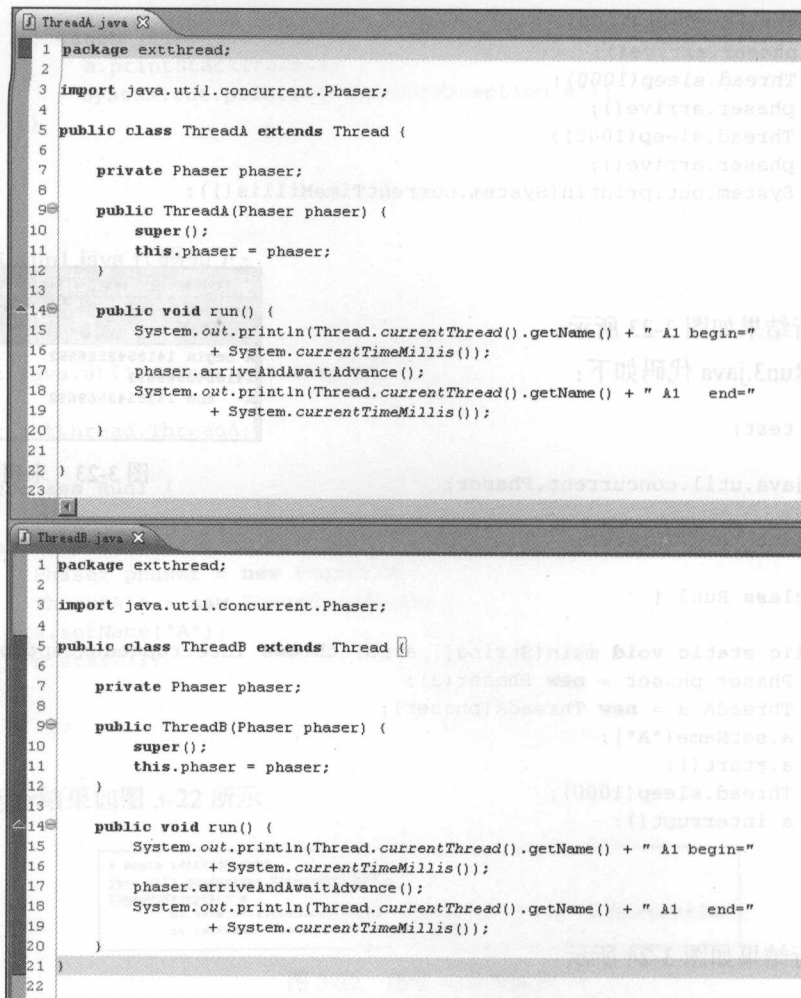


图 3-25 线程类代码

运行类 Run1.java 代码如下：

```

package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;
import extthread.ThreadB;

public class Run1 {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(3);
        ThreadA a = new ThreadA(phaser);
        a.setName("A");
    }
}

```

```

a.start();
ThreadB b = new ThreadB(phaser);
b.setName("B");
b.start();
}
}

```

程序运行结果如图 3-26 所示。

控制台出现了 2 个 begin, 说明 2 个线程呈阻塞状态, 因为计数未达到 3。

运行类 Run2.java 代码如下:

```

package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;
import extthread.ThreadB;

public class Run2 {
    public static void main(String[] args) {
        try {
            Phaser phaser = new Phaser(3);
            ThreadA a = new ThreadA(phaser);
            a.setName("A");
            a.start();
            ThreadB b = new ThreadB(phaser);
            b.setName("B");
            b.start();
            Thread.sleep(1000);
            phaser.forceTermination();
            System.out.println(phaser.isTerminated());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 3-27 所示。

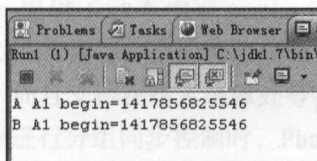


图 3-26 运行结果

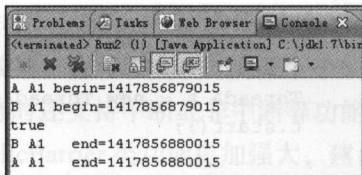


图 3-27 运行结果

从控制台打印的结果来看, 类 Phaser 执行 forceTermination() 方法时仅仅将屏障取消, 线程继续执行后面的代码, 并不出现异常, 而 CyclicBarrier 类的 reset() 方法执行时却出现异常。

3.17 控制 Phaser 类的运行时机

前面章节的实验都是线程一起到达屏障后继续运行，有些情况下是需要进行控制的，也就是到达屏障后不允许继续运行。

创建项目 Phaser_testA，类代码如下：

```
package extthread;

import java.util.concurrent.Phaser;

public class ThreadA extends Thread {

    private Phaser phaser;

    public ThreadA(Phaser phaser) {
        super();
        this.phaser = phaser;
    }

    public void run() {
        System.out.println(Thread.currentThread().getName() + " A1 begin="
            + System.currentTimeMillis());
        phaser.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getName() + " A1 end="
            + System.currentTimeMillis());
    }
}
```

创建 Run1.java 文件，代码如下：

```
package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;

public class Run1 {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(3);
        for (int i = 0; i < 3; i++) {
            ThreadA t = new ThreadA(phaser);
            t.start();
        }
    }
}
```

程序运行结果如图 3-28 所示。

创建新的运行类 Run2.java 代码如下：

```

package test;

import java.util.concurrent.Phaser;

import extthread.ThreadA;

public class Run2 {
    public static void main(String[] args) throws InterruptedException {
        Phaser phaser = new Phaser(3);
        phaser.register();
        for (int i = 0; i < 3; i++) {
            ThreadA t = new ThreadA(phaser);
            t.start();
        }
        Thread.sleep(5000);
        phaser.arriveAndDeregister();
    }
}

```

运行结果如图 3-29 所示。

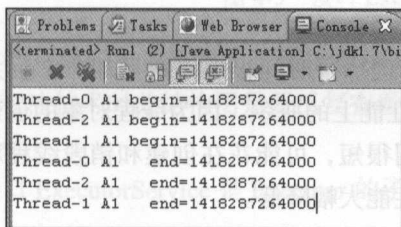


图 3-28 默认的运行效果

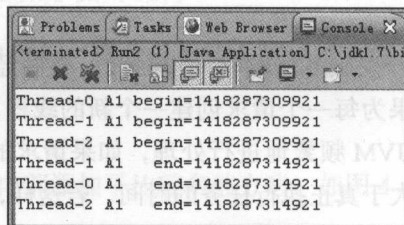


图 3-29 运行时机可控了

此实验说明运行的时机是可以通过逻辑控制的，主要的原理就是计数 +1，然后通过逻辑代码的方式来决定线程是否继续向下运行。

3.18 本章总结

类 Phaser 提供了动态增减 parties 计数，这点比 CyclicBarrier 类操作 parties 更加方便，通过若干个方法来控制多个线程之间同步运行的效果，还可以实现针对某一个线程取消同步运行的效果，而且支持在指定屏障处等待，在等待时还支持中断或非中断等功能，使用 Java 并发类对线程进行分组同步控制时，Phaser 比 CyclicBarrier 类功能更加强大，建议使用。

public class Thread extends Thread {

时间大于真正执行任务的时间,若这样,则系统性能大幅降低。

部分类都是实现此接口的，该接口声明如图 4-1 所示。

此接口的结构非常简洁，仅有一个方法，如图 4-2 所示。

java.util.concurrent

接口 Executor

所有已知子接口：

ExecutorService, ScheduledExecutorService

所有已知实现类：

AbstractExecutorService, ScheduledThreadPoolExecutor, ThreadPoolExecutor

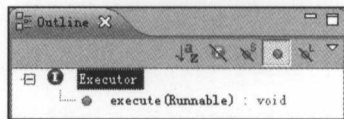


图 4-1 接口 Executor 的声明

图 4-2 仅有 1 个 execute() 方法

但 Executor 是接口, 并不能直接使用, 所以还得需要实现类, 图 4-3 中所示的内容就是完整的 Executor 接口相关的类继承结构。

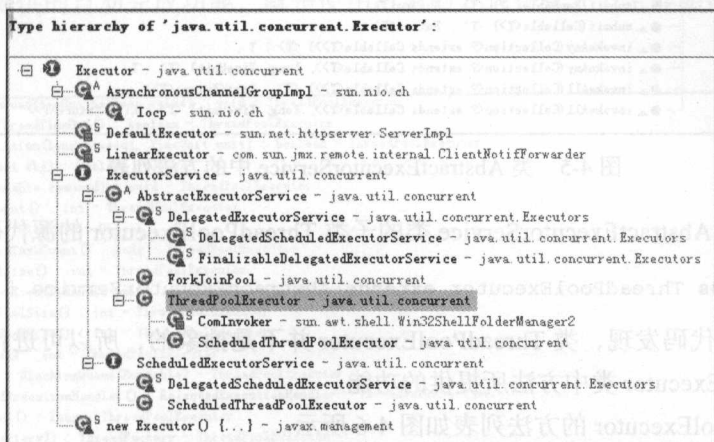


图 4-3 接口 Executor 的完整实现与继承结构

这幅结构图相当重要, 它概括了从 Executor 祖先接口到实现类的全部实现与继承过程, 学习线程池技术时此结构图是要反复查看的。

继续!

接口 ExecutorService 是 Executor 的子接口, 在内部添加了比较多的方法, 如图 4-4 所示。

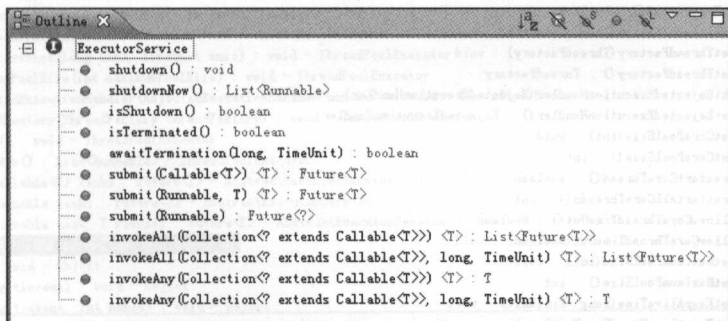


图 4-4 接口 ExecutorService 的内部结构

虽然接口 ExecutorService 添加了若干个方法的定义, 但还是不能实例化, 那么就要看一下它的唯一子实现类 AbstractExecutorService, AbstractExecutorService 类中的方法列表如图 4-5 所示。

根据类 AbstractExecutorService 的名称来看, 它是 abstract 抽象的, 查看源代码如下, 也的确是抽象类:

```
public abstract class AbstractExecutorService implements ExecutorService {
```

所以类 AbstractExecutorService 同样也是不能实例化的。

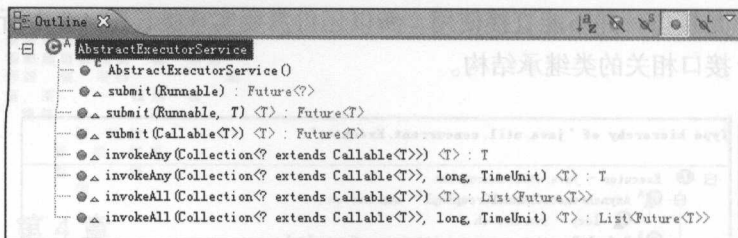


图 4-5 类 AbstractExecutorService 中的方法列表

再来看一下 AbstractExecutorService 类的子类 ThreadPoolExecutor 的源代码如下：

```
public class ThreadPoolExecutor extends AbstractExecutorService {
```

通过查看源代码发现，类 ThreadPoolExecutor 并不是抽象的，所以可进行实例化，进而使用 ThreadPoolExecutor 类中方法所提供的功能。

类 ThreadPoolExecutor 的方法列表如图 4-6 所示。

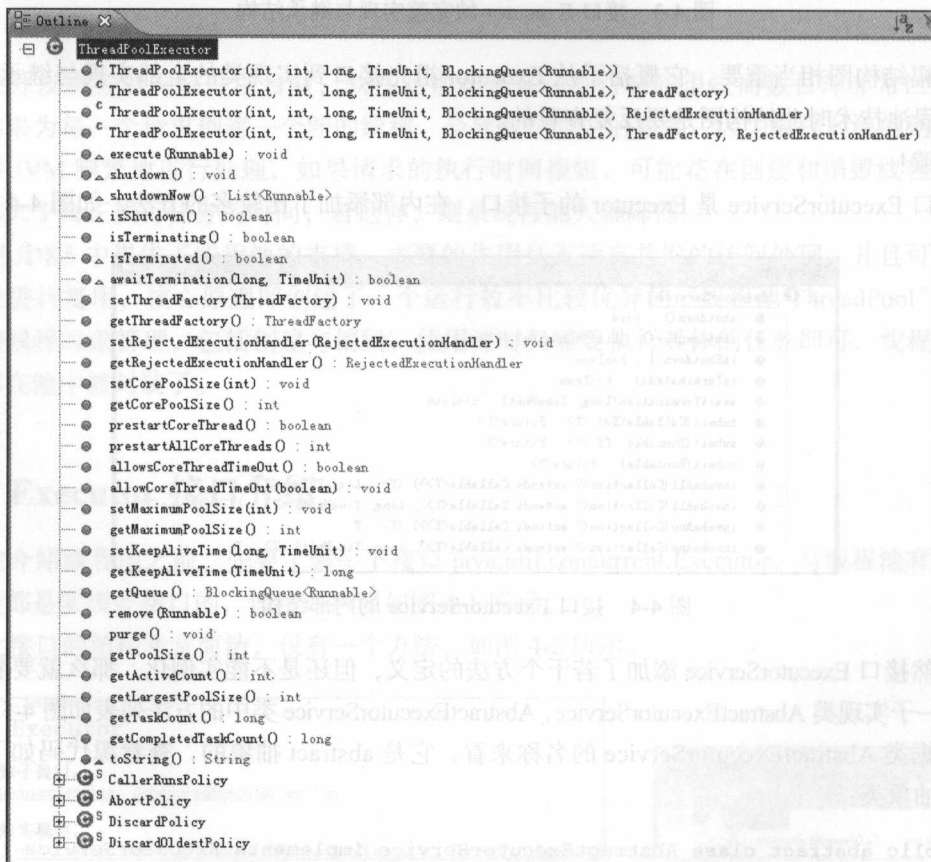


图 4-6 类 ThreadPoolExecutor 的方法列表

图 4-6 所提供的信息是 ThreadPoolExecutor 类中方法的列表，并未显示从父类继承而又没有重写的方法，为了查看 ThreadPoolExecutor 对象能调用的全部方法列表，声明一个变量，然后通过 IDE 提供的自动完成功能，就是使用对象的方式查看一下全部能调用的方法列表，如图 4-7 所示。

```

allowCoreThreadTimeOut(boolean value) : void - ThreadPoolExecutor
allowsCoreThreadTimeOut() : boolean - ThreadPoolExecutor
awaitTermination(long timeout, TimeUnit unit) : boolean - ThreadPoolExecutor
equals(Object obj) : boolean - Object
execute(Runnable command) : void - ThreadPoolExecutor
getActiveCount() : int - ThreadPoolExecutor
getClass() : Class<?> - Object
getCompletedTaskCount() : long - ThreadPoolExecutor
getCorePoolSize() : int - ThreadPoolExecutor
getKeepAliveTime(TimeUnit unit) : long - ThreadPoolExecutor
getLargestPoolSize() : int - ThreadPoolExecutor
getMaximumPoolSize() : int - ThreadPoolExecutor
getPoolSize() : int - ThreadPoolExecutor
getQueue() : BlockingQueue<Runnable> - ThreadPoolExecutor
getRejectedExecutionHandler() : RejectedExecutionHandler - ThreadPoolExecutor
getTaskCount() : long - ThreadPoolExecutor
getThreadFactory() : ThreadFactory - ThreadPoolExecutor
hashCode() : int - Object
invokeAll(Collection<? extends Callable<T>> tasks) : List<Future<T>> - AbstractExecutorService
invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) : List<Future<T>> - AbstractExecutorService
invokeAny(Collection<? extends Callable<T>> tasks) : T - AbstractExecutorService
invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) : T - AbstractExecutorService
isShutdown() : boolean - ThreadPoolExecutor
isTerminated() : boolean - ThreadPoolExecutor
isTerminating() : boolean - ThreadPoolExecutor
notify() : void - Object
notifyAll() : void - Object
prestartAllCoreThreads() : int - ThreadPoolExecutor
prestartCoreThread() : boolean - ThreadPoolExecutor
purge() : void - ThreadPoolExecutor
remove(Runnable task) : boolean - ThreadPoolExecutor
setCorePoolSize(int corePoolSize) : void - ThreadPoolExecutor
setKeepAliveTime(long time, TimeUnit unit) : void - ThreadPoolExecutor
setMaximumPoolSize(int maximumPoolSize) : void - ThreadPoolExecutor
setRejectedExecutionHandler(RejectedExecutionHandler handler) : void - ThreadPoolExecutor
setThreadFactory(ThreadFactory threadFactory) : void - ThreadPoolExecutor
shutdown() : void - ThreadPoolExecutor
shutdownNow() : List<Runnable> - ThreadPoolExecutor
submit(Callable<T> task) : Future<T> - AbstractExecutorService
submit(Runnable task) : Future<?> - AbstractExecutorService
submit(Runnable task, T result) : Future<T> - AbstractExecutorService
toString() : String - ThreadPoolExecutor
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object

```

图 4-7 类 ThreadPoolExecutor 的对象所能调用的全部方法列表

4.2 节和 4.3 节就是介绍这些方法的使用，从而可以对线程池所提供的功能了解得更加全面。

4.2 使用 Executors 工厂类创建线程池

接口 Executor 仅仅是一种规范，是一种声明，是一种定义，并没有实现任何的功能，所

以大多数的情况下，需要使用接口的实现类来完成指定的功能，比如 `ThreadPoolExecutor` 类就是 `Executor` 的实现类，但 `ThreadPoolExecutor` 在使用上并不是那么方便，在实例化时需要传入很多个参数，还要考虑线程的并发数等与线程池运行效率有关的参数，所以官方建议使用 `Executors` 工厂类来创建线程池对象。

`Executors` 工厂类的结构如图 4-8 所示。

类 `Executors` 中的方法如图 4-9 所示。

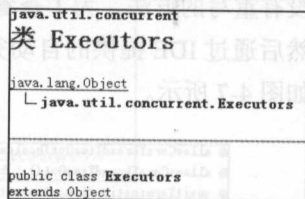


图 4-8 类 `Executors` 结构

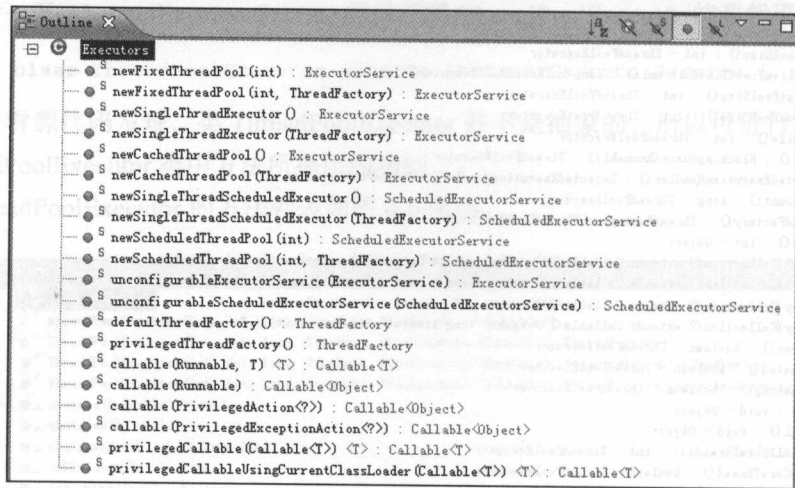


图 4-9 类 `Executors` 中的方法

在下面的章节将实验几个比较常用的 API。

4.2.1 使用 `newCachedThreadPool()` 方法创建无界线程池

使用 `Executors` 类的 `newCachedThreadPool()` 方法创建的是无界线程池，可以进行线程自动回收。所谓的“无界线程池”就是池中存放线程个数是理论上的 `Integer.MAX_VALUE` 最大值。

创建实验用的项目 `Executors_1`，类 `Run1.java` 代码如下：

```

package test.run;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Run1 {

    public static void main(String[] args) {

        ExecutorService executorService = Executors.newCachedThreadPool();
    }
}

```

```

import java.util.concurrent.*;

public class Runnable1 {
    @Override
    public void run() {
        try {
            System.out.println("Runnable1 begin "
                               + System.currentTimeMillis());
            Thread.sleep(1000);
            System.out.println("A");
            System.out.println("Runnable1 end "
                               + System.currentTimeMillis());
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

```

import java.util.concurrent.*;

public class Runnable2 {
    @Override
    public void run() {
        try {
            System.out.println("Runnable2 begin "
                               + System.currentTimeMillis());
            Thread.sleep(1000);
            System.out.println("B");
            System.out.println("Runnable2 end "
                               + System.currentTimeMillis());
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

程序运行后的效果如图 4-10 所示。

从打印的时间来看，A 和 B 几乎是在相同的时间开始 begin 的，也就是创建了 2 个线程，2 个线程之间是异步运行的。

继续实验，创建新的类 Run2.java，代码如下：

```

package test.run;

import java.util.concurrent.*;

public class Run2 {

    public static void main(String[] args) {

```

```

        ExecutorService executorService = Executors.newCachedThreadPool();

```

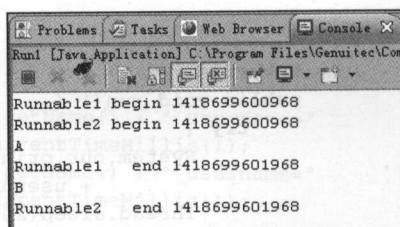


图 4-10 打印了 AB

```

以大多数的情况
for (int i = 0; i < 5; i++) {
    executorService.execute(new Runnable() {
        @Override
        public void run() {
            System.out.println("run!");
        }
    });
}
}
}

```

程序运行结果如图 4-11 所示。

循环打印也成功。

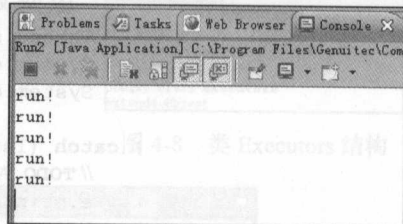


图 4-11 循环创建 Runnable 对象

4.2.2 验证 newCachedThreadPool() 创建为 Thread 池

前面的实验都没有验证 newCachedThreadPool() 方法创建的是线程池，在本测试中将要到论证。

创建项目 Executors_2，类 MyRunnable.java 代码如下：

```

package myrunnable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " username="
                + username + " begin " + System.currentTimeMillis());
            Thread.sleep(2000);
            System.out.println(Thread.currentThread().getName() + " username="
                + username + " end " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

运行类 Run.java 代码如下：

```

package test.run;

import java.util.concurrent.ExecutorService;

```

```

import java.util.concurrent.Executors;

import myrunnable.MyRunnable;

public class Run {

    public static void main(String[] args) {

        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
            executorService.execute(new MyRunnable((" " + (i + 1))));
        }
    }
}

```

程序运行后如图 4-12 所示。

说明线程池对象创建是完全成功的，但还没有达到池中线程对象可以复用的效果，下面要实现这样的效果。

创建新的项目 Executors_2_1，创建 MyRunnable.java 类代码如下：

```

package myrunnable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " username="
            + username + " begin " + System.currentTimeMillis());
        System.out.println(Thread.currentThread().getName() + " username="
            + username + " end " + System.currentTimeMillis());
    }
}

```

运行类 Run.java 代码如下：

```

package test.run;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import myrunnable.MyRunnable;

public class Run {

```



```

public static void main(String[] args) throws InterruptedException {

    ExecutorService executorService = Executors.newCachedThreadPool();
    for (int i = 0; i < 5; i++) {
        executorService.execute(new MyRunnable(("" + (i + 1))));
    }
    Thread.sleep(1000);
    System.out.println("");
    System.out.println("");
    for (int i = 0; i < 5; i++) {
        executorService.execute(new MyRunnable(("" + (i + 1))));
    }
}

```

程序运行结果如图 4-13 所示。

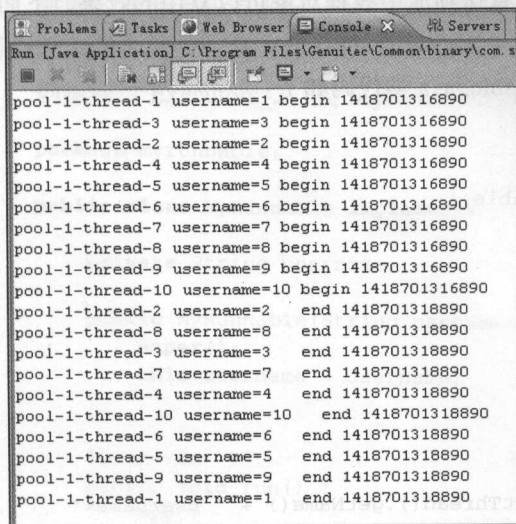


图 4-12 池中创建了 10 个线程

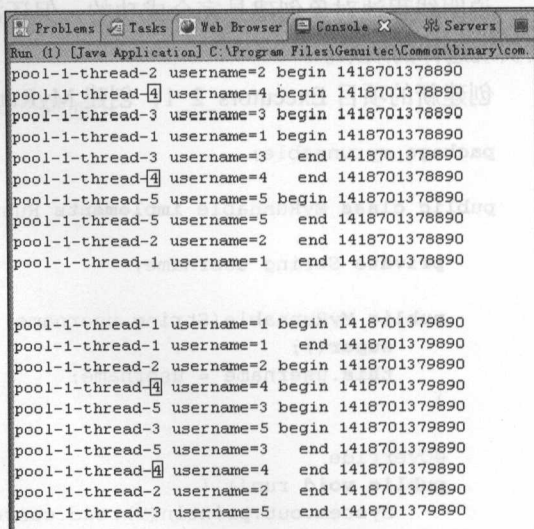


图 4-13 复用线程对象了

4.2.3 使用 newCachedThreadPool (ThreadFactory) 定制线程工厂

无界线程池中的 Thread 类还可以由程序员自己定制，方法 newCachedThreadPool(ThreadFactory) 就是解决这个问题的。

创建项目 newCachedThreadPoolFactory，创建 MyThreadFactory.java 线程工厂类代码如下：

```

package mythreadfactory;

import java.util.concurrent.ThreadFactory;

public class MyThreadFactory implements ThreadFactory {

```

```

public Thread newThread(Runnable r) {
    Thread thread = new Thread(r);
    thread.setName("定制池中的线程对象的名称" + Math.random());
    return thread;
}
}

```

运行类 Run.java 代码如下:

```

package test;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import mythreadfactory.MyThreadFactory;

public class Run {
    public static void main(String[] args) {
        MyThreadFactory threadFactory = new MyThreadFactory();
        ExecutorService executorService = Executors
            .newCachedThreadPool(threadFactory);
        executorService.execute(new Runnable() {
            public void run() {
                System.out.println("我在运行" + System.currentTimeMillis() + " "
                    + Thread.currentThread().getName());
            }
        });
    }
}

```

程序运行结果如图 4-14 所示。

通过使用自定义的 ThreadFactory 接口实

我在运行1431935423937 定制池中的线程对象的名称0.3657112342080461

现类, 实现了线程对象的定制性。

图 4-14 运行结果

4.2.4 使用 newFixedThreadPool(int) 方法创建有界线程池

方法 newFixedThreadPool(int) 创建的是有界线程池, 也就是池中的线程个数可以指定最大数量。

创建项目 Executors_3, 类 MyRunnable.java 代码如下:

```

package myrunnable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    @Override
    public void run() {

```

```

    try {
        System.out.println(Thread.currentThread().getName() + " username="
            + username + " begin " + System.currentTimeMillis());
        Thread.sleep(2000);
        System.out.println(Thread.currentThread().getName() + " username="
            + username + " end " + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

运行类 Run.java 代码如下：

```

package test.run;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import myrunnable.MyRunnable;

public class Run {

    public static void main(String[] args) {

        ExecutorService executorService = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 3; i++) {
            executorService.execute(new MyRunnable((" " + (i + 1))));
        }
        for (int i = 0; i < 3; i++) {
            executorService.execute(new MyRunnable((" " + (i + 1))));
        }
    }
}

```

程序运行后的效果如图 4-15 所示。

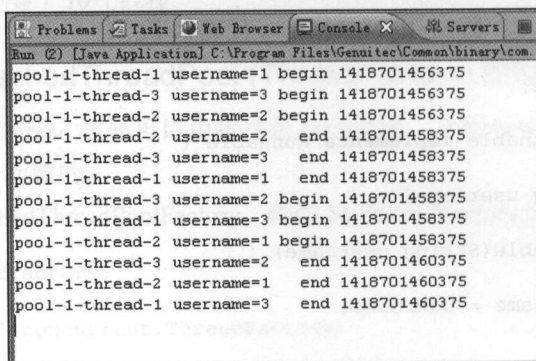


图 4-15 最多只有 3 个线程在运行

使用有界线程池后线程池中的最多线程个数是可控的。

4.2.5 使用 newFixedThreadPool(int, ThreadFactory) 定制线程工厂

有界线程池中的 Thread 类还可以由程序员自己定制, 方法 newFixedThreadPool(int nThreads, ThreadFactory threadFactory) 就是解决这个问题。

创建项目 newFixedThreadPoolFactory, 创建 MyThreadFactory.java 线程工厂类代码如下:

```
package mythreadfactory;

import java.util.concurrent.ThreadFactory;

public class MyThreadFactory implements ThreadFactory {

    public Thread newThread(Runnable r) {
        Thread thread = new Thread(r);
        thread.setName("定制池中的线程对象的名称" + Math.random());
        return thread;
    }
}
```

运行类 Run.java 代码如下:

```
package test;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import mythreadfactory.MyThreadFactory;

public class Run {
    public static void main(String[] args) {
        MyThreadFactory threadFactory = new MyThreadFactory();
        ExecutorService executorService = Executors.newFixedThreadPool(2,
            threadFactory);
        Runnable runnable = new Runnable() {
            public void run() {
                try {
                    System.out.println("begin 我在运行 "
                        + System.currentTimeMillis() + " "
                        + Thread.currentThread().getName());
                    Thread.sleep(3000);
                    System.out.println("end 我在运行 "
                        + System.currentTimeMillis() + " "
                        + Thread.currentThread().getName());
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        executorService.execute(runnable);
    }
}
```

```

        executorService.execute(runnable);
        executorService.execute(runnable);
    }
}

```

程序运行结果如图 4-16 所示。

同一时间只有 2 个线程在池中被运行，因为使用的是大小为 2 的有界线程池。

begin	我在运行1431935962703	定制池中的线程对象的名称0.4753577878028954
begin	我在运行1431935962703	定制池中的线程对象的名称0.9064607141221289
end	我在运行1431935965703	定制池中的线程对象的名称0.4753577878028954
end	我在运行1431935965703	定制池中的线程对象的名称0.9064607141221289
begin	我在运行1431935965703	定制池中的线程对象的名称0.4753577878028954
end	我在运行1431935968703	定制池中的线程对象的名称0.4753577878028954

图 4-16 运行结果

4.2.6 使用 newSingleThreadExecutor() 方法创建单一线程池

使用 newSingleThreadExecutor() 方法可以创建单一线程池，单一线程池可以实现以队列的方式来执行任务。

创建项目 Executors_4，类 MyRunnable.java 代码如下：

```

package myrunnable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " username="
                + username + " begin " + System.currentTimeMillis());
            Thread.sleep(2000);
            System.out.println(Thread.currentThread().getName() + " username="
                + username + " end " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

运行类 Run.java 代码如下：

```

package test.run;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import myrunnable.MyRunnable;

public class Run {

```

```

public static void main(String[] args) {
    ExecutorService executorService = Executors.newSingleThreadExecutor();
    for (int i = 0; i < 3; i++) {
        executorService.execute(new MyRunnable(("" + (i + 1))));
    }
}

```

程序运行后的效果如图 4-17 所示。

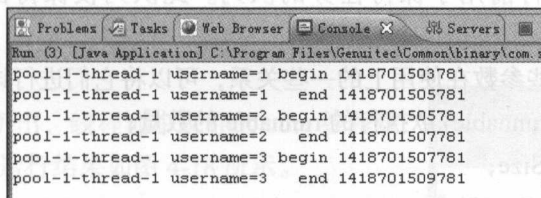


图 4-17 最多只有 1 个线程在运行

4.2.7 使用 newSingleThreadExecutor(ThreadFactory) 定制线程工厂

此方法的使用方式与前面章节的使用方式大体一致，在此不重复演示。

4.3 ThreadPoolExecutor 的使用

类 ThreadPoolExecutor 可以非常方便地创建线程池对象，而不需要程序员设计大量的 new 实例化 Thread 相关的代码。

4.2 节使用 Executors 工厂类的 newXXXThreadExecutor() 方法可以快速方便地创建线程池，但创建的细节却未知，通过查看源代码在调用 newSingleThreadExecutor() 方法时内部其实是实例化了 1 个 ThreadPoolExecutor 类的实例，源代码如下：

```

public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}

```

所以下一步就要细化研究一下 ThreadPoolExecutor 类的使用。

4.3.1 构造方法的测试

类 ThreadPoolExecutor 最常使用的构造方法是：

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)

参数解释如下：

- ❑ `corePoolSize`：池中所保存的线程数，包括空闲线程，也就是核心池的大小。
- ❑ `maximumPoolSize`：池中允许的最大线程数。
- ❑ `keepAliveTime`：当线程数量大于 `corePoolSize` 值时，在没有超过指定的时间内是不从线程池中删除空闲线程的，如果超过此时间单位，则删除。
- ❑ `unit`：`keepAliveTime` 参数的时间单位。
- ❑ `workQueue`：执行前用于保持任务的队列。此队列仅保持由 `execute` 方法提交的

4.2.6 Runnable 任务。

为了更好地理解这些参数在使用上的一些关系，可以将它们进行详细化的注释：

- 1) A 代表 `execute(runnable)` 欲执行的 `runnable` 的数量；
- 2) B 代表 `corePoolSize`；
- 3) C 代表 `maximumPoolSize`；
- 4) D 代表 `A-B` (假设 $A \geq B$)；
- 5) E 代表 `new LinkedBlockingDeque<Runnable>()`；队列，无构造参数；
- 6) F 代表 `SynchronousQueue` 队列；
- 7) G 代表 `keepAliveTime`。

构造方法中 5 个参数之间都有使用上的关系，在使用线程池的过程中大部分会出现如下 5 种过程：

<A> 如果 $A \leq B$ ，那么马上创建线程运行这个任务，并不放入扩展队列 `Queue` 中，其他参数功能忽略；

 如果 $A > B \&\& A \leq C \&\& E$ ，则 C 和 G 参数忽略，并把 D 放入 E 中等待被执行；

<C> 如果 $A > B \&\& A \leq C \&\& F$ ，则 C 和 G 参数有效，并且马上创建线程运行这些任务，而不把 D 放入 F 中，D 执行完任务后在指定时间后发生超时时将 D 进行清除；

<D> 如果 $A > B \&\& A > C \&\& E$ ，则 C 和 G 参数忽略，并把 D 放入 E 中等待被执行；

<E> 如果 $A > B \&\& A > C \&\& F$ ，则处理 C 的任务，其他任务则不再处理抛出异常。

在下面的章节将分为若干情况进行实验。

1. 前 2 个参数与 `getCorePoolSize()` 和 `getMaximumPoolSize()` 方法

创建实验用的项目，名称为 `ThreadPoolExecutor_1`，创建类 `Run1.java` 代码如下：

```
package test.run;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run1 {
    // 获取基本属性 corePoolSize 和 maximumPoolSize
```

```

public static void main(String[] args) {
    ThreadPoolExecutor executor = new ThreadPoolExecutor(7, 8, 5,
        TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
    System.out.println(executor.getCorePoolSize());
    System.out.println(executor.getMaximumPoolSize());
    System.out.println("");
    executor = new ThreadPoolExecutor(7, 8, 5, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>());
    System.out.println(executor.getCorePoolSize());
    System.out.println(executor.getMaximumPoolSize());
}

```

从代码中可以分析出，线程池中保存的 core 线程数是 7，最大为 8，程序运行结果如图 4-18 所示。

2. 在线程池中添加的线程数量 \leq corePoolSize

前一个实验并未在池中创建线程，继续实验。

创建类 Run2_1.java 代码如下：

```
package test.run;
```

```
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
```

```
public class Run2_1 {
```

```
    // 队列使用 LinkedBlockingDeque 类
```

```
    // 并且线程数量  $\leq$  corePoolSize
```

```
    // 所以 keepAliveTime>5 时也不清除空闲线程
```

```
    public static void main(String[] args) throws InterruptedException {
```

```
        Runnable runnable = new Runnable() {
```

```
            @Override
```

```
            public void run() {
```

```
                try {
```

```
                    System.out.println(Thread.currentThread().getName()
```

```
                        + " run!" + System.currentTimeMillis());
```

```
                    Thread.sleep(1000);
```

```
                } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
```

```
                }
```

```
            }
        }
    }
}

```

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(7, 8, 5,
```

```
    TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
```

```
executor.execute(runnable); // 1
```

```
executor.execute(runnable); // 2
```

```
executor.execute(runnable); // 3
```

```
executor.execute(runnable); // 4
```

```
executor.execute(runnable); // 5
```

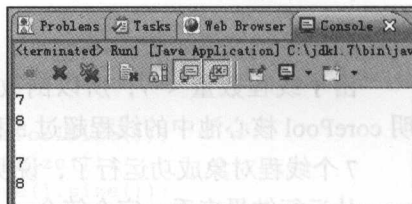


图 4-18 运行结果

```

        executor.execute(runnable);// 6
        executor.execute(runnable);// 7
        Thread.sleep(300);
        System.out.println("A:" + executor.getCorePoolSize());
        System.out.println("A:" + executor.getPoolSize());
        System.out.println("A:" + executor.getQueue().size());
        Thread.sleep(10000);
        System.out.println("B:" + executor.getCorePoolSize());
        System.out.println("B:" + executor.getPoolSize());
        System.out.println("B:" + executor.getQueue().size());
    }
    // 按钮呈红色, 因为池中还有线程在等待任务
}

```

程序运行结果如图 4-19 所示。

由于线程数量 ≤ 7 , 所以倒数第 2 个打印为 7, 说明 corePool 核心池中的线程超过 5 秒钟不清除。

7 个线程对象成功运行了, 说明线程池成功工作了。

从运行结果来看, 完全符合:

<A> 如果 $A \leq B$, 那么马上创建线程运行这个任务, 并不放入扩展队列 Queue 中, 其他参数功能忽略。

可以将下面 4 个方法做一个比喻, 便于理解:

```

// 车中可载人的标准人数
System.out.println(pool.getCorePoolSize());
// 车中可载人的最大人数
System.out.println(pool.getMaximumPoolSize());
// 车中正在载的人数
System.out.println(pool.getPoolSize());
// 扩展车中正在载的人数
System.out.println(pool.getQueue().size());

```

创建类 Run2_2.java 代码如下:

```

package test.run;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run2_2 {
    // 队列使用 SynchronousQueue 类
    // 并且线程数量  $\leq$  corePoolSize
    // 所以 keepAliveTime > 5 时也不清除空闲线程
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName())

```

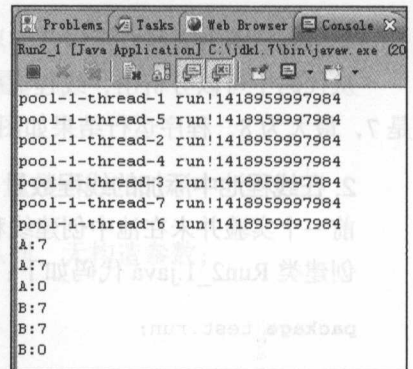


图 4-19 运行结果

```

        + " run!" + System.currentTimeMillis());
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

ThreadPoolExecutor executor = new ThreadPoolExecutor(7, 8, 5,
    TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
executor.execute(runnable); // 1
executor.execute(runnable); // 2
executor.execute(runnable); // 3
executor.execute(runnable); // 4
executor.execute(runnable); // 5
executor.execute(runnable); // 6
executor.execute(runnable); // 7
Thread.sleep(300);
System.out.println("A:" + executor.getCorePoolSize());
System.out.println("A:" + executor.getPoolSize());
System.out.println("A:" + executor.getQueue().size());
Thread.sleep(10000);
System.out.println("B:" + executor.getCorePoolSize());
System.out.println("B:" + executor.getPoolSize());
System.out.println("B:" + executor.getQueue().size());
}

```

// 按钮呈红色, 因为池中还有线程在等待任务

程序运行结果如图 4-20 所示。

7 个线程对象成功运行了, 说明线程池成功工作了。

从运行结果来看, 完全符合:

<A> 如果 $A \leq B$, 那么马上创建线程运行这个任务, 并不放入扩展队列 Queue 中, 其他参数功能忽略。

3. 数量 $> \text{corePoolSize}$ 并且 $\leq \text{maximumPoolSize}$ 的情况

创建 Run3_1.java 类, 代码如下:

```

package test.run;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run3_1 {
    // 队列使用 LinkedBlockingDeque 类, 也就是如果
    // 线程数量 > corePoolSize 时将其余的任务放入队列中
    // 同一时间最多只能有 7 个线程在运行
    // 如果使用 LinkedBlockingDeque 类则 maximumPoolSize 参数作用将忽略
    public static void main(String[] args) throws InterruptedException {

```

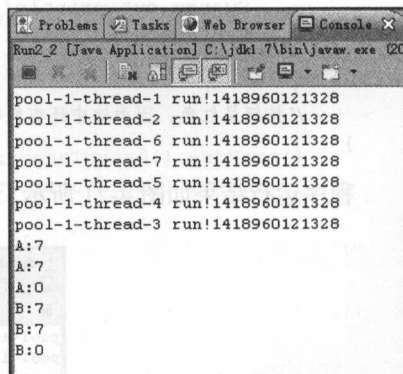


图 4-20 运行结果

```

Runnable runnable = new Runnable() {
    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName()
                + " run!" + System.currentTimeMillis());
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};

ThreadPoolExecutor executor = new ThreadPoolExecutor(7, 8, 5,
    TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
executor.execute(runnable);// 1
executor.execute(runnable);// 2
executor.execute(runnable);// 3
executor.execute(runnable);// 4
executor.execute(runnable);// 5
executor.execute(runnable);// 6
executor.execute(runnable);// 7
executor.execute(runnable);// 8
Thread.sleep(300);
System.out.println("A:" + executor.getCorePoolSize());
System.out.println("A:" + executor.getPoolSize());
System.out.println("A:" + executor.getQueue().size());
Thread.sleep(10000);
System.out.println("B:" + executor.getCorePoolSize());
System.out.println("B:" + executor.getPoolSize());
System.out.println("B:" + executor.getQueue().size());
}
// 按钮呈红色, 因为池中还有线程在等待任务
}

```

程序运行结果如图 4-21 所示。

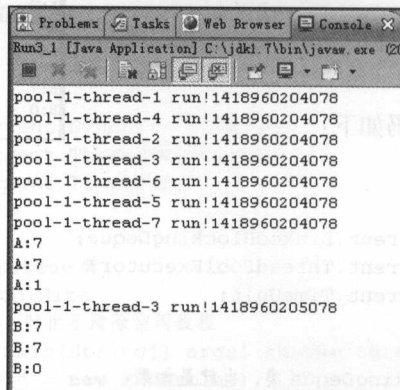


图 4-21 运行结果

8 个线程成功运行。

证明以下结论是成立的:

 如果 $A > B$ 且 $A \leq C$ 且 E , 则 C 和 G 参数忽略, 并把 D 放入 E 中等待被执行

BlockingQueue 只是一个接口, 常用的实现类有 LinkedBlockingQueue 和 ArrayBlockingQueue。用 LinkedBlockingQueue 的好处在于没有大小限制, 优点是队列容量非常大, 所以执行 execute() 不会抛出异常, 而线程池中运行的线程数也永远不会超过 corePoolSize 值, 因为其他多余的线程被放入 LinkedBlockingQueue 队列中, keepAliveTime 参数也就没有意义了。

创建 Run3_2.java 类, 代码如下:

```
package test.run;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run3_2 {
    // 队列使用 SynchronousQueue 类
    // 并且线程数量 > corePoolSize 时
    // 将其余的任务也放入池中, 总数量为 8,
    // 并且线程总数量也没有超过 maximumPoolSize 值的 8
    // 由于运行的线程数为 8, 数量上 > corePoolSize 为 7 的值
    // 所以 keepAliveTime > 5 时清除空闲线程
    // 如果使用 SynchronousQueue 类则 maximumPoolSize 参数的作用将有效
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName()
                        + " run!" + System.currentTimeMillis());
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(7, 8, 5,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
        executor.execute(runnable); // 1
        executor.execute(runnable); // 2
        executor.execute(runnable); // 3
        executor.execute(runnable); // 4
        executor.execute(runnable); // 5
        executor.execute(runnable); // 6
        executor.execute(runnable); // 7
        executor.execute(runnable); // 8
        Thread.sleep(300);
        System.out.println("A:" + executor.getCorePoolSize());
        System.out.println("A:" + executor.getPoolSize());
    }
}
```



```

        System.out.println("A:" + executor.getQueue().size());
        Thread.sleep(10000);
        System.out.println("B:" + executor.getCorePoolSize());
        System.out.println("B:" + executor.getPoolSize());
        System.out.println("B:" + executor.getQueue().size());
    }

```

// 按钮呈红色, 因为池中还有线程在等待任务

// 删除的是 >corePoolSize 的多余线程

程序运行结果如图 4-22 所示:

8 个线程成功运行。

证明以下结论是成立的:

<C> 如果 $A > B$ 且 $A \leq C$ 且 F , 则 C 和 G 参数有效, 并且马上创建线程运行这些任务, 而不把 D 放入 F 中, D 执行完任务后在指定时间后发生超时时将 D 进行清除

4. 数量 >maximumPoolSize 的情况

创建 Run4_1.java 类, 代码如下:

```

package test.run;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run4_1 {
    // 队列使用 LinkedBlockingDeque 类
    // 并且线程数量 >corePoolSize 时将其余的任务放入队列中
    // 同一时间最多只能有 corePoolSize 个线程在运行
    // 所以 keepAliveTime>5 时也不清除空闲线程
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName()
                        + " run!" + System.currentTimeMillis());
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(7, 8, 5,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        executor.execute(runnable); // 1
        executor.execute(runnable); // 2
        executor.execute(runnable); // 3
        executor.execute(runnable); // 4
    }
}

```

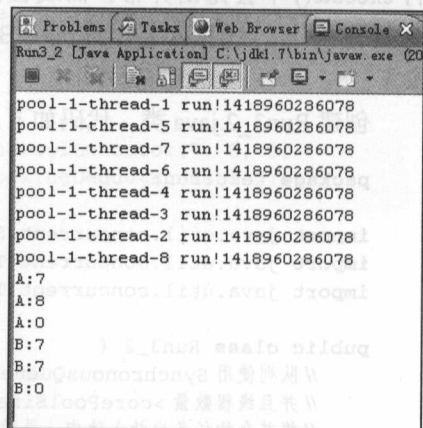


图 4-22 运行结果

```

    executor.execute(runnable); // 5
    executor.execute(runnable); // 6
    executor.execute(runnable); // 7
    executor.execute(runnable); // 8
    executor.execute(runnable); // 9
    Thread.sleep(300);
    System.out.println("A:" + executor.getCorePoolSize());
    System.out.println("A:" + executor.getPoolSize());
    System.out.println("A:" + executor.getQueue().size());
    Thread.sleep(10000);
    System.out.println("B:" + executor.getCorePoolSize());
    System.out.println("B:" + executor.getPoolSize());
    System.out.println("B:" + executor.getQueue().size());
}
}

```

程序运行结果如图 4-23 所示。

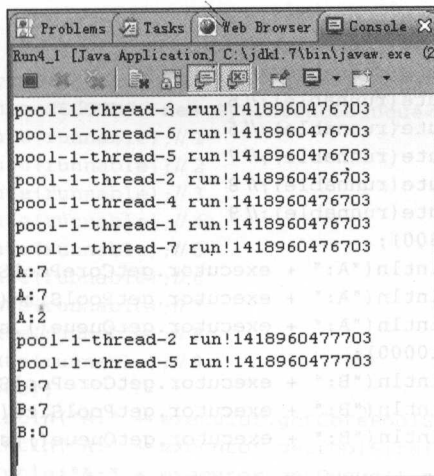


图 4-23 不报错的情况

证明以下结论是成立的：

<D> 如果 $A > B \&\& A > C \&\& E$ ，则 C 和 G 参数忽略，并把 D 放入 E 中等待被执行
创建 Run4_2.java 类，代码如下：

```

package test.run;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run4_2 {
    // 队列使用 SynchronousQueue 类
    // 线程数量 >= corePoolSize

```

```

// 并且线程数量 <= maximumPoolSize
// 所以 keepAliveTime > 5 时清除空闲线程
public static void main(String[] args) throws InterruptedException {
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName()
                    + " run!" + System.currentTimeMillis());
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    };

    ThreadPoolExecutor executor = new ThreadPoolExecutor(7, 10, 5,
        TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
    executor.execute(runnable); // 1
    executor.execute(runnable); // 2
    executor.execute(runnable); // 3
    executor.execute(runnable); // 4
    executor.execute(runnable); // 5
    executor.execute(runnable); // 6
    executor.execute(runnable); // 7
    executor.execute(runnable); // 8
    executor.execute(runnable); // 9
    Thread.sleep(300);
    System.out.println("A:" + executor.getCorePoolSize());
    System.out.println("A:" + executor.getPoolSize());
    System.out.println("A:" + executor.getQueue().size());
    Thread.sleep(10000);
    System.out.println("B:" + executor.getCorePoolSize());
    System.out.println("B:" + executor.getPoolSize());
    System.out.println("B:" + executor.getQueue().size());
}
// 按钮呈红色，因为池中还有线程在等待任务
}

```

程序运行结果如图 4-24 所示。

从运行结果来看也是符合：

<C> 如果 $A > B \& \& A \leq C \& \& F$ ，则 C 和 G 参数有效，并且马上创建线程运行这些任务，而不把 D 放入 F 中，D 执行完任务后在指定时间后发生超时时将 D 进行清除。

创建 Run4_3.java 类，代码如下：

```

package test.run;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;

```

```

pool-1-thread-3 run!1418960547187
pool-1-thread-1 run!1418960547187
pool-1-thread-6 run!1418960547187
pool-1-thread-5 run!1418960547187
pool-1-thread-4 run!1418960547187
pool-1-thread-7 run!1418960547187
pool-1-thread-8 run!1418960547187
pool-1-thread-2 run!1418960547187
pool-1-thread-9 run!1418960547187
A:7
A:9
A:0
B:7
B:7
B:0

```

图 4-24 运行结果

```

import java.util.concurrent.TimeUnit;

public class Run4_3 {
    // 队列使用 SynchronousQueue 类
    // 线程数量 > corePoolSize
    // 并且线程数量 > maximumPoolSize
    // 所以出现异常
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName()
                        + " run!" + System.currentTimeMillis());
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        ThreadPoolExecutor executor = new ThreadPoolExecutor(7, 8, 5,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
        executor.execute(runnable); // 1
        executor.execute(runnable); // 2
        executor.execute(runnable); // 3
        executor.execute(runnable); // 4
        executor.execute(runnable); // 5
        executor.execute(runnable); // 6
        executor.execute(runnable); // 7
        executor.execute(runnable); // 8
        executor.execute(runnable); // 9
        Thread.sleep(300);
        System.out.println("A:" + executor.getCorePoolSize());
        System.out.println("A:" + executor.getPoolSize());
        System.out.println("A:" + executor.getQueue().size());
        Thread.sleep(10000);
        System.out.println("B:" + executor.getCorePoolSize());
        System.out.println("B:" + executor.getPoolSize());
        System.out.println("B:" + executor.getQueue().size());
    }
    // 按钮会变灰
}

```

程序运行结果如下：

```

pool-1-thread-1 run!1418960811734
pool-1-thread-4 run!1418960811734
pool-1-thread-2 run!1418960811734
pool-1-thread-3 run!1418960811734
pool-1-thread-6 run!1418960811734
pool-1-thread-5 run!1418960811734
pool-1-thread-7 run!1418960811734
pool-1-thread-8 run!1418960811734

```

```
Exception in thread "main"
java.util.concurrent.RejectedExecutionException: Task
test.run.Run4_3$1@1ee2c2c rejected from
java.util.concurrent.ThreadPoolExecutor@1963b3e[Running, pool size = 8,
active threads = 8, queued tasks = 0, completed tasks = 0]
    at
    java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution
(ThreadPoolExecutor.java:2048)
    at
    java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:821)
    at
    java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1372)
    at test.run.Run4_3.main(Run4_3.java:35)
```

证明以下结论是成立的：

<E> 如果 $A > B \&\& A > C \&\& F$ ，则处理 C 的任务，其他任务则不再处理抛出异常

5. 参数 keepAliveTime 为 0 时的实验

keepAliveTime：当线程数量大于 **corePoolSize** 值时，在没有超过指定的时间内是不从线程池中删除空闲线程的，如果超过此时间单位，则删除，如果为 0 则任务执行完毕后立即从队列中删除。

unit：keepAliveTime 参数的时间单位。

创建 Run5.java 文件，代码如下：

```
package test.run;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run5 {
    // 队列使用 SynchronousQueue 类
    // 线程数量 > corePoolSize
    // 并且线程数量 <= maximumPoolSize
    // 并且 keepAliveTime 值为 0 时的作用是线程执行完后立即清除
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName()
                        + " run!" + System.currentTimeMillis());
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
    }
}
```

```

ThreadPoolExecutor executor = new ThreadPoolExecutor(7, 10, 0L,
    TimeUnit.MILLISECONDS, new SynchronousQueue<Runnable>());
executor.execute(runnable);//1
executor.execute(runnable);//2
executor.execute(runnable);//3
executor.execute(runnable);//4
executor.execute(runnable);//5
executor.execute(runnable);//6
executor.execute(runnable);//7
executor.execute(runnable);//8
executor.execute(runnable);//9
Thread.sleep(300);
System.out.println("A:" + executor.getCorePoolSize());
System.out.println("A:" + executor.getPoolSize());
System.out.println("A:" + executor.getQueue().size());
Thread.sleep(5000);
System.out.println("B:" + executor.getCorePoolSize());
System.out.println("B:" + executor.getPoolSize());
System.out.println("B:" + executor.getQueue().size());
}
}

```

程序运行后的效果如图 4-25 所示。

倒数第 2 个 B 由 9 变成 7 是因为超时时间设置为 0L，线程执行完毕后立即将空闲的线程从非 corePool 中删除，而 corePool 的数量还是 7。

4.3.2 方法 shutdown() 和 shutdownNow() 与返回值

方法 shutdown() 的作用是使当前未执行完的线程继续执行，而不再添加新的任务 Task，还有 shutdown() 方法不会阻塞，调用 shutdown() 方法后，主线程 main 就马上结束了，而线程池会继续运行直到所有任务执行完才会停止。如果不调用 shutdown() 方法，那么线程池会一直保持下去，以便随时执行被添加的新 Task 任务。

方法 shutdownNow() 的作用是中断所有的任务 Task，并且抛出 InterruptedException 异常，前提是在 Runnable 中使用 if (Thread.currentThread().isInterrupted() == true) 语句来判断当前线程的中断状态，而未执行的线程不再执行，也就是从执行队列中清除。如果没有 if (Thread.currentThread().isInterrupted() == true) 语句及抛出异常的代码，则池中正在运行的线程直到执行完毕，而未执行的线程不再执行，也从执行队列中清除。

创建测试用的项目 ThreadPoolExecutor_2，类 MyRunnable1.java 代码如下：

```

package myrunnable;

public class MyRunnable1 implements Runnable {

```

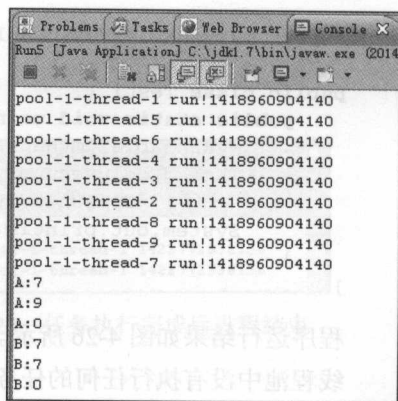


图 4-25 运行结果


```

public void run() {
    try {
        System.out.println("begin " + Thread.currentThread().getName()
            + " " + System.currentTimeMillis());
        Thread.sleep(4000);
        System.out.println("end " + Thread.currentThread().getName()
            + " " + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

类 Test1.java 代码如下:

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(7, 10, 0L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        System.out.println("main end!");
    }
}

```

程序运行结果如图 4-26 所示。

线程池中并没有执行任何的任务, 继续实验。

类 Test2.java 代码如下:

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(7, 10, 0L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(myRunnable);
        System.out.println("main end!");
    }
}

```

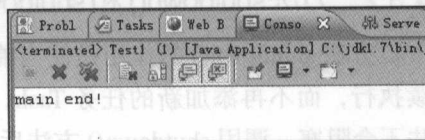


图 4-26 无 Task 任务被执行进程结束

程序运行结果如图 4-27 所示。

类 Test3.java 代码如下:

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test3 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(7, 10, 0L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(myRunnable);
        pool.shutdown();
        System.out.println("main end!");
    }
}
```

程序运行结果如图 4-28 所示。

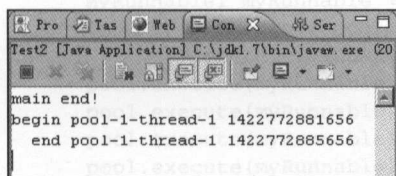


图 4-27 任务执行完成后池继续等待新的任务

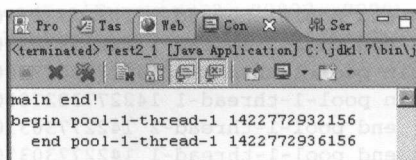


图 4-28 任务执行完成后进程结束

程序运行的效果是 4 秒钟之后进程结束。

类 Test4.java 代码如下:

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test4 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
    }
}
```

```

        pool.execute(myRunnable);
        Thread.sleep(1000);
        pool.shutdown();
        pool.execute(myRunnable);
        System.out.println("main end!");
    }
}

```

程序运行结果如下：

```

begin pool-1-thread-1 1422773023000
begin pool-1-thread-2 1422773023000
Exception in thread "main"
java.util.concurrent.RejectedExecutionException: Task
myRunnable.MyRunnable1@1be16f5 rejected from
java.util.concurrent.ThreadPoolExecutor@d56b37[Shutting down, pool
size = 2, active threads = 2, queued tasks = 2, completed tasks = 0]
    at
    java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoo
lExecutor.java:2048)
    at
    java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:821)
    at
    java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1372)
        at test.Test4.main(Test4.java:20)
    end pool-1-thread-2 1422773027000
    end pool-1-thread-1 1422773027000
begin pool-1-thread-2 1422773027000
begin pool-1-thread-1 1422773027000
    end pool-1-thread-2 1422773031000
    end pool-1-thread-1 1422773031000

```

通过运行结果可知，执行了 4 个任务，最后一个任务报异常，因为执行了 shutdown() 方法，并且当前程序进程销毁。

继续学习 shutdownNow() 方法。

创建测试用的项目 ThreadPoolExecutor_2_shutdownNow，类 MyRunnable1.java 代码如下：

```

package myRunnable;

public class MyRunnable1 implements Runnable {
    public void run() {
        try {
            for (int i = 0; i < Integer.MAX_VALUE / 50; i++) {
                String newString = new String();
                Math.random();
                Math.random();
                Math.random();
                Math.random();
                Math.random();
                Math.random();

                if (Thread.currentThread().isInterrupted() == true) {

```

```

        System.out.println(" 任务没有完成, 就中断了! ");
        throw new InterruptedException();
    }
}

    System.out.println(" 任务成功完成! ");
} catch (InterruptedException e) {
    System.out.println(" 进入 catch 中断了任务 ");
    e.printStackTrace();
}
}
}

```

类 Test1.java 代码如下:

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        Thread.sleep(1000);
        pool.shutdownNow();
    }
}

```

程序运行结果如下:

任务没有完成, 就中断了!

```

java.lang.InterruptedException
    at myrunnable.MyRunnable1.run(MyRunnable1.java:17)
    at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)
java.lang.InterruptedException
    at myrunnable.MyRunnable1.run(MyRunnable1.java:17)
    at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)

```

任务没有完成，就中断了！

进入 catch 中断了任务

进入 catch 中断了任务

从控制台输出的信息来看，剩下的 2 个任务被取消，没有被运行。

类 MyRunnable2.java 代码如下：

```
package myrunnable;

public class MyRunnable2 implements Runnable {
    public void run() {
        for (int i = 0; i < Integer.MAX_VALUE / 1000; i++) {
            new String();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
        }
        System.out.println("任务成功完成！");
    }
}
```

类 Test2.java 代码如下：

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable2;

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable2 myRunnable = new MyRunnable2();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        Thread.sleep(1000);
        pool.shutdownNow();
        System.out.println("main end!");
    }
}
```

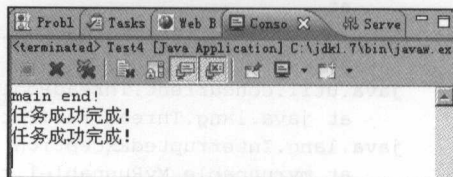


图 4-29 运行结果

程序运行结果如图 4-29 所示。

控制台信息代表 2 个任务被成功执行，其余 2 个任务被取消运行，并且进程销毁。

类 Test3.java 代码如下:

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable2;

public class Test3 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable2 myRunnable = new MyRunnable2();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        Thread.sleep(1000);
        pool.shutdownNow();
        pool.execute(myRunnable);
        System.out.println("main end!");
    }
}
```

程序运行结果如下:

```
Exception in thread "main"
java.util.concurrent.RejectedExecutionException: Task
myrunnable.MyRunnable2@1947496 rejected from
java.util.concurrent.ThreadPoolExecutor@1724a9d[Shutting down, pool
size = 2, active threads = 2, queued tasks = 0, completed tasks = 0]
    at
java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution
(ThreadPoolExecutor.java:2048)
    at
java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:821)
    at
java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1372)
    at test.Test3.main(Test3.java:20)
任务成功完成!
任务成功完成!
```

控制台信息代表 2 个任务被成功执行, 其余 2 个任务被取消运行, 而最后一个任务则拒绝执行, 运行效果即抛出异常, 并且在最后进程销毁。

当线程池调用 shutdown() 方法时, 线程池的状态则立刻变成 SHUTDOWN 状态, 此时不能再往线程池中添加任何任务, 否则将会抛出 RejectedExecutionException 异常。但是, 此时线程池不会立刻退出, 直到线程池中的任务都已经处理完成, 才会退出。

而 `shutdownNow()` 方法是使线程池的状态立刻变成 STOP 状态，并试图停止所有正在执行的线程（如果有 if 判断则人为地抛出异常），不再处理还在池队列中等待的任务，当然，它会返回那些未执行的任务。

另外，在调用 `shutdown()` 方法后，正在执行的任务和队列中的任务在后期正常执行，验证一下此结论，创建测试用的项目 test26，类 `MyRunnableA.java` 代码如下：

```
package myrunnable;

public class MyRunnableA implements Runnable {

    private String username;

    public MyRunnableA(String username) {
        super();
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    @Override
    public void run() {
        for (int i = 0; i < Integer.MAX_VALUE / 500; i++) {
            String newString1 = new String();
            String newString5 = new String();
            String newString6 = new String();
            String newString7 = new String();
            Math.random();
            Math.random();
            Math.random();
        }
        System.out.println(Thread.currentThread().getName() + " 任务完成！");
    }
}
```

类 `Test1.java` 代码如下：

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnableA;

public class Test1 {

    public static void main(String[] args) {
        try {
```

```

MyRunnableA a1 = new MyRunnableA("A1");
MyRunnableA a2 = new MyRunnableA("A2");
MyRunnableA a3 = new MyRunnableA("A3");
MyRunnableA a4 = new MyRunnableA("A4");

ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 10, 30,
    TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
pool.execute(a1);
pool.execute(a2);
pool.execute(a3);
pool.execute(a4);

Thread.sleep(1000);

pool.shutdown();

System.out.println("main end!");
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

```

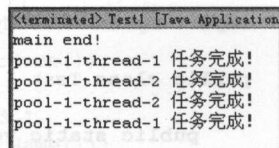


图 4-30 全部任务被成功执行

程序运行结果如图 4-30 所示。

而在调用 `shutdownNow()` 方法后队列中的任务被取消运行，方法 `shutdownNow()` 的返回值是 `List<Runnable>`，`List` 对象存储的是还未运行的任务，也就是被取消掉的任务。为了验证存储的是未运行的任务，创建实验用的项目 `test27`，类 `MyRunnable.java` 代码如下：

```

package myrunnable;

public class MyRunnableA implements Runnable {

    private String username;

    public MyRunnableA(String username) {
        super();
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    @Override
    public void run() {
        for (int i = 0; i < Integer.MAX_VALUE / 500; i++) {
            String newString1 = new String();
            String newString5 = new String();
            String newString6 = new String();
            String newString7 = new String();
            Math.random();

```

```

        Math.random();
        Math.random();
    }
    System.out.println(Thread.currentThread().getName() + " 任务完成! ");
}
}
}

```

类 Test.java 代码如下:

```

package test;

import java.util.List;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnableA;

public class Test {

    public static void main(String[] args) {
        try {
            MyRunnableA a1 = new MyRunnableA("A1");
            MyRunnableA a2 = new MyRunnableA("A2");
            MyRunnableA a3 = new MyRunnableA("A3");
            MyRunnableA a4 = new MyRunnableA("A4");

            ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 10, 30,
                TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
            pool.execute(a1);
            pool.execute(a2);
            pool.execute(a3);
            pool.execute(a4);

            Thread.sleep(1000);

            List<Runnable> list = pool.shutdownNow();

            for (int i = 0; i < list.size(); i++) {
                MyRunnableA myRunnableA = (MyRunnableA) list.get(i);
                System.out.println(myRunnableA.getUsername() + " 任务被取消! ");
            }

            System.out.println("main end!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 4-31 所示。

图 4-31 有两个任务被取消了

```

<terminated> Test (7) [Java Applicati...
A3 任务被取消!
A4 任务被取消!
main end!
pool-1-thread-1 任务完成!
pool-1-thread-2 任务完成!

```

4.3.3 方法 isShutdown()

方法 isShutdown() 的作用是判断线程池是否已经关闭。

创建测试用的项目 ThreadPoolExecutor_3, 类 Run1.java 代码如下:

```
package test.run;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run1 {
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println("打印了! begin "
                        + Thread.currentThread().getName());
                    Thread.sleep(1000);
                    System.out.println("打印了! end "
                        + Thread.currentThread().getName());
                } catch (InterruptedException e) {
                    //TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 2,
            Integer.MAX_VALUE, TimeUnit.SECONDS,
            new LinkedBlockingDeque<Runnable>());
        executor.execute(runnable);
        System.out.println("A=" + executor.isShutdown());
        executor.shutdown();
        System.out.println("B=" + executor.isShutdown());
    }
}
```

程序运行结果如图 4-32 所示。

只要调用了 shutdown() 方法, isShutdown() 方法的返回值就是 true。

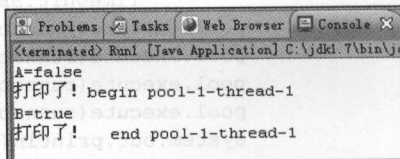


图 4-32 运行结果

4.3.4 方法 isTerminating() 和 isTerminated()

如果正在执行的程序处于 shutdown 或 shutdownNow 之后处于正在终止但尚未完全终止的过程中, 调用方法 isTerminating() 则返回 true。此方法可以比喻成, 门是否正在关闭。门彻底关闭时, 线程池也就关闭了。

如果线程池关闭后，也就是所有任务都已完成，则方法 `isTerminated()` 返回 `true`。此方法可以比喻成，门是否已经关闭。

创建测试用的项目 `ThreadPoolExecutor_4`，创建类 `MyRunnable.java` 代码如下：

```
package myrunnable;

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " begin "
                + System.currentTimeMillis());
            Thread.sleep(2000);
            System.out.println(Thread.currentThread().getName() + " end "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 `Test.java` 代码如下：

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class Test {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable runnable = new MyRunnable();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 99999,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(runnable);
        pool.execute(runnable);
        pool.execute(runnable);
        pool.execute(runnable);
        System.out.println(pool.isTerminating() + " " + pool.isTerminated());
        pool.shutdown();
        Thread.sleep(1000);
        System.out.println(pool.isTerminating() + " " + pool.isTerminated());
        Thread.sleep(1000);
        System.out.println(pool.isTerminating() + " " + pool.isTerminated());
        Thread.sleep(1000);
        System.out.println(pool.isTerminating() + " " + pool.isTerminated());
        Thread.sleep(1000);
        System.out.println(pool.isTerminating() + " " + pool.isTerminated());
    }
}
```

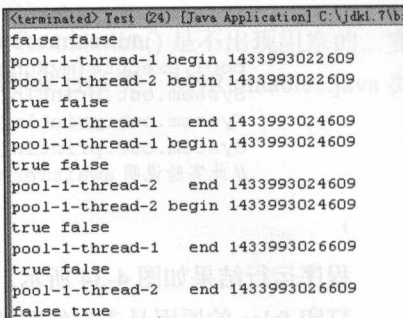
```

        Thread.sleep(1000);
        System.out.println(pool.isTerminating()
            + " " + pool.isTerminated());
    }
}

```

程序运行结果如图 4-33 所示。

方法 shutdown() 或 shutdownNow() 的功能是发出一个关闭大门的命令，方法 isShutdown() 是判断这个关闭大门的命令发出或未发出。方法 isTerminating() 代表大门是否正在关闭进行中，而 isTerminated() 方法判断大门是否已经关闭了。



```

<terminated> Test (24) [Java Application] C:\jdk1.7\bin
false false
pool-1-thread-1 begin 1433993022609
pool-1-thread-2 begin 1433993022609
true false
pool-1-thread-1 end 1433993024609
pool-1-thread-1 begin 1433993024609
true false
pool-1-thread-2 end 1433993024609
pool-1-thread-2 begin 1433993024609
true false
pool-1-thread-1 end 1433993026609
true false
pool-1-thread-2 end 1433993026609
false true

```

图 4-33 运行结果

4.3.5 方法 awaitTermination(long timeout, TimeUnit unit)

方法 awaitTermination(long timeout, TimeUnit unit) 的作用就是查看在指定的时间之间，线程池是否已经终止工作，也就是最多等待多少时间去判断线程池是否已经终止工作。

此方法需要有 shutdown() 方法的配合。

创建测试用的项目 ThreadPoolExecutor_5，创建类 MyRunnable1.java 代码如下：

```

package myrunnable;

public class MyRunnable1 implements Runnable {
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " "
                + System.currentTimeMillis());
            Thread.sleep(4000);
            System.out.println(Thread.currentThread().getName() + " "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

类 Test1.java 代码如下：

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {

```



```

MyRunnable1 myRunnable = new MyRunnable1();
ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
    TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
pool.execute(myRunnable);
System.out.println("main begin ! " + System.currentTimeMillis());
System.out.println(pool.awaitTermination(10, TimeUnit.SECONDS));
System.out.println("main end ! " + System.currentTimeMillis());
// 此实验说明 awaitTermination() 方法具有阻塞特性
}

```

程序运行结果如图 4-34 所示。

打印 false 的原因是未对线程池执行 shutdown() 方法。

类 Test2.java 代码如下：

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(myRunnable);
        pool.shutdown();
        System.out.println("main begin ! " + System.currentTimeMillis());
        System.out.println(pool.awaitTermination(10, TimeUnit.SECONDS));
        System.out.println("main end ! " + System.currentTimeMillis());
        // 代码: awaitTermination(10, TimeUnit.SECONDS) 作用:
        // 最多等待 10 秒, 也就是阻塞 10 秒
    }
}

```

程序运行结果如图 4-35 所示。

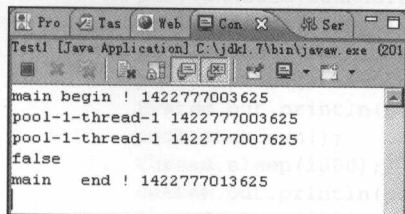


图 4-34 有阻塞的特性

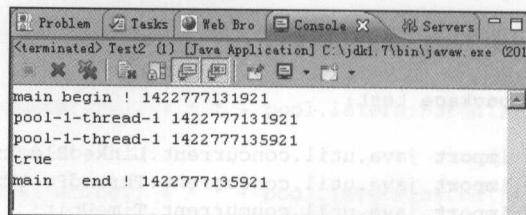


图 4-35 打印结果为 true

从打印的结果来看，“main end !”打印的时间就是线程执行完毕后的时间，也就是说方法 awaitTermination() 被执行时，如果池中有任务在被执行时，则调用 awaitTermination() 方

法出现阻塞,等待指定的时间,如果没有任务时则不再阻塞。

下面验证一下如果池中没有任何任务要执行时,方法 `awaitTermination()` 是不出现阻塞的。实验代码在 `Test3.java` 中,文件 `Test3.java` 需要使用 `MyRunnable2.java` 类,`MyRunnable2.java` 类代码如下:

```
package myrunnable;

public class MyRunnable2 implements Runnable {
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 `Test3.java` 代码如下:

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable2;

public class Test3 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable2 myRunnable = new MyRunnable2();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(1, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.shutdown();
        System.out.println("A=" + pool.awaitTermination(1, TimeUnit.SECONDS)
            + " " + System.currentTimeMillis());
        System.out.println("B=" + pool.awaitTermination(1, TimeUnit.SECONDS)
            + " " + System.currentTimeMillis());
        System.out.println("C=" + pool.awaitTermination(1, TimeUnit.SECONDS)
            + " " + System.currentTimeMillis());
        System.out.println("D=" + pool.awaitTermination(1, TimeUnit.SECONDS)
            + " " + System.currentTimeMillis());
        System.out.println("E=" + pool.awaitTermination(1, TimeUnit.SECONDS)
            + " " + System.currentTimeMillis());
        System.out.println("F=" + pool.awaitTermination(1, TimeUnit.SECONDS)
            + " " + System.currentTimeMillis());
        System.out.println("G=" + pool.awaitTermination(1, TimeUnit.SECONDS)
            + " " + System.currentTimeMillis());
    }
}
```

程序运行结果如图 4-36 所示。

方法 `awaitTermination()` 与 `shutdown()` 结合时可以实现“等待执行完毕”的效果，原理就是应用 `awaitTermination()` 方法具有阻塞性，如果 `awaitTermination()` 方法正在阻塞的过程中任务执行完毕，则 `awaitTermination()` 取消阻塞继续执行后面的代码，此实验在 `Test4.java` 文件中，代码如下：

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test4 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.execute(myRunnable);
        pool.shutdown();
        System.out.println(pool.awaitTermination(Integer.MAX_VALUE,
            TimeUnit.SECONDS)
            + " " + System.currentTimeMillis() + " 全部任务执行完毕!");
    }
}
```

程序运行后的效果如图 4-37 所示。

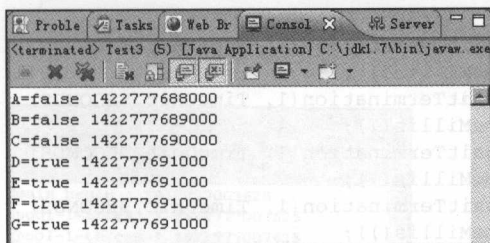


图 4-36 运行结果

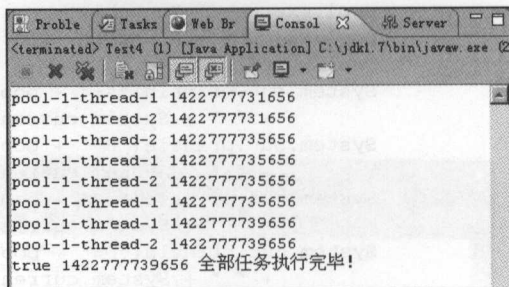


图 4-37 等待运行结束

4.3.6 工厂 `ThreadFactory+execute()+UncaughtExceptionHandler` 处理异常

有时需要对线程池中创建的线程属性进行定制化，这时就得需要配置 `ThreadFactory` 线程工厂。

创建测试用的项目 ThreadPoolExecutor_6, 类 MyRunnable1.java 代码如下:

```
package myrunnable;

public class MyRunnable1 implements Runnable {
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " "
                + System.currentTimeMillis());
            Thread.sleep(4000);
            System.out.println(Thread.currentThread().getName() + " "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 MyThreadFactoryA.java 代码如下:

```
package mythreadfactory;

import java.util.Date;
import java.util.concurrent.ThreadFactory;

public class MyThreadFactoryA implements ThreadFactory {
    public Thread newThread(Runnable r) {
        Thread newThread = new Thread(r);
        newThread.setName("高洪岩: " + new Date());
        return newThread;
    }
}
```

类 Test1.java 代码如下:

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;
import mythreadfactory.MyThreadFactoryA;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>(),
            new MyThreadFactoryA());
        pool.execute(myRunnable);
    }
}
```

程序运行结果如图 4-38 所示。

除了使用构造方法传递自定义 ThreadFactory 外，还可以使用 setThreadFactory() 方法来设置自定义 ThreadFactory，实验代码在 Test2.java 文件中，代码如下：

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;
import mythreadfactory.MyThreadFactoryA;

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.setThreadFactory(new MyThreadFactoryA());
        pool.execute(myRunnable);
    }
}
```

程序运行结果如图 4-39 所示。

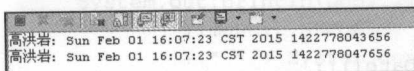


图 4-38 运行结果

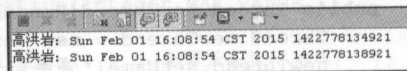


图 4-39 运行结果

当线程运行出现异常时，则 JDK 抛出异常，此实验需要使用 MyRunnable2.java 类，代码如下：

```
package myrunnable;

public class MyRunnable2 implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " "
            + System.currentTimeMillis());
        String abc = null;
        abc.indexOf(0);
        System.out.println(Thread.currentThread().getName() + " "
            + System.currentTimeMillis());
    }
}
```

文件 Test3.java 代码如下：

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
```

```

import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable2;
import mythreadfactory.MyThreadFactoryA;

public class Test3 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable2 myRunnable = new MyRunnable2();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.setThreadFactory(new MyThreadFactoryA());
        pool.execute(myRunnable);
    }
}

```

程序运行后的效果如图 4-40 所示。

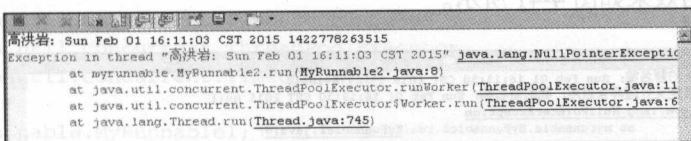


图 4-40 程序员无法自行处理异常

在使用自定义线程工厂时，线程如果出现异常完全可以自定义处理的，需要创建 MyThreadFactoryB.java，代码如下：

```

package mythreadfactory;

import java.lang.Thread.UncaughtExceptionHandler;
import java.util.Date;
import java.util.concurrent.ThreadFactory;

public class MyThreadFactoryB implements ThreadFactory {
    public Thread newThread(Runnable r) {
        Thread newThread = new Thread(r);
        newThread.setName("我的新名称：" + new Date());
        newThread.setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
            public void uncaughtException(Thread t, Throwable e) {
                System.out.println("自定义处理异常启用：" + t.getName() + " "
                    + e.getMessage());
                e.printStackTrace();
            }
        });
        return newThread;
    }
}

```

创建 Test4.java 文件，代码如下：

```

package test;

```



```

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable2;
import mythreadfactory.MyThreadFactoryB;

public class Test4 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable2 myRunnable = new MyRunnable2();
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 99999, 9999L,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        pool.setThreadFactory(new MyThreadFactoryB());
        pool.execute(myRunnable);
    }
}

```

程序运行后的效果如图 4-41 所示。

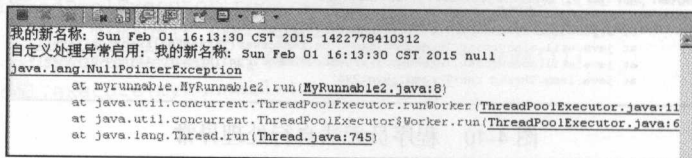


图 4-41 成功捕捉异常信息

4.3.7 方法 set/getRejectedExecutionHandler()

方法 setRejectedExecutionHandler() 和 getRejectedExecutionHandler() 的作用是可以处理任务被拒绝执行时的行为。

创建测试用的项目 ThreadPoolExecutor_7，创建 MyRunnable1.java 类代码如下：

```

package myrunnable;

public class MyRunnable1 implements Runnable {
    private String username;

    public MyRunnable1(String username) {
        super();
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}

```

```

public void run() {
    try {
        System.out.println(Thread.currentThread().getName() + " "
            + System.currentTimeMillis());
        Thread.sleep(4000);
        System.out.println(Thread.currentThread().getName() + " "
            + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

类 Test1.java 代码如下:

```

package test;

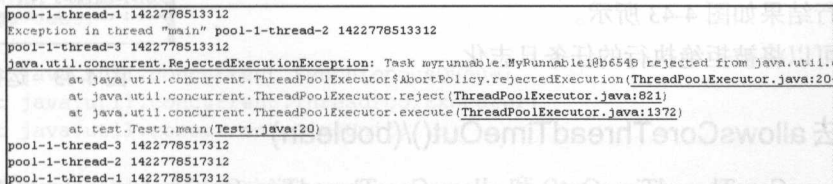
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable1 = new MyRunnable1(" 中国 1");
        MyRunnable1 myRunnable2 = new MyRunnable1(" 中国 2");
        MyRunnable1 myRunnable3 = new MyRunnable1(" 中国 3");
        MyRunnable1 myRunnable4 = new MyRunnable1(" 中国 4");
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 9999L,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
        pool.execute(myRunnable1);
        pool.execute(myRunnable2);
        pool.execute(myRunnable3);
        pool.execute(myRunnable4);
    }
}

```

程序运行结果如图 4-42 所示。



```

pool-1-thread-1 1422778513312
Exception in thread "main" pool-1-thread-2 1422778513312
pool-1-thread-3 1422778513312
java.util.concurrent.RejectedExecutionException: Task myrunnable.MyRunnable1@b6548 rejected from java.util.
    at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:20
    at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:821)
    at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1372)
    at test.Test1.main(Test1.java:20)
pool-1-thread-3 1422778517312
pool-1-thread-2 1422778517312
pool-1-thread-1 1422778517312

```

图 4-42 拒绝运行多余的任务

在出现这样的异常时可以自定义拒绝执行任务的行为, 创建类 MyRejectedExecution Handler.java 代码如下:

```

package myrejectedexectionhandler;

import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadPoolExecutor;

import myrunnable.MyRunnable1;

public class MyRejectedExecutionHandler implements RejectedExecutionHandler {
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        System.out.println(((MyRunnable1) r).getUsername() + " 被拒绝执行");
    }
}

```

类 Test2.java 代码如下：

```

package test;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrejectedexectionhandler.MyRejectedExecutionHandler;
import myrunnable.MyRunnable1;

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable1 = new MyRunnable1(" 中国 1");
        MyRunnable1 myRunnable2 = new MyRunnable1(" 中国 2");
        MyRunnable1 myRunnable3 = new MyRunnable1(" 中国 3");
        MyRunnable1 myRunnable4 = new MyRunnable1(" 中国 4");

        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 9999L,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
        pool.setRejectedExecutionHandler(new MyRejectedExecutionHandler());
        pool.execute(myRunnable1);
        pool.execute(myRunnable2);
        pool.execute(myRunnable3);
        pool.execute(myRunnable4);
    }
}

```

程序运行结果如图 4-43 所示。

此实验可以将被拒绝执行的任务日志化。

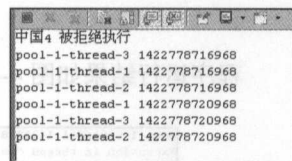


图 4-43 运行结果

4.3.8 方法 allowsCoreThreadTimeOut()/(boolean)

方法 allowsCoreThreadTimeOut() 和 allowsCoreThreadTimeOut(boolean value) 的作用是配置核心线程是否有超时的效果。

创建测试用的项目 ThreadPoolExecutor_8，创建类 MyRunnable.java 代码如下：

```

package myrunnable;

```

```

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " begin "
            + System.currentTimeMillis());
        System.out.println(Thread.currentThread().getName() + " end "
            + System.currentTimeMillis());
    }
}

```

类 Test1.java 代码如下:

```

package test;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class Test1 {

    public static void main(String[] args) throws InterruptedException {
        ThreadPoolExecutor pool = new ThreadPoolExecutor(4, 5, 5,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
        System.out.println(pool.allowsCoreThreadTimeOut());
        for (int i = 0; i < 4; i++) {
            MyRunnable runnable = new MyRunnable();
            pool.execute(runnable);
        }
        Thread.sleep(8000);
        System.out.println(pool.getPoolSize());
    }
}

```

程序运行结果如图 4-44 所示。

创建类 Test2.java 代码如下:

```

package test;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class Test2 {

    public static void main(String[] args) throws InterruptedException {
        ThreadPoolExecutor pool = new ThreadPoolExecutor(4, 5, 5,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
    }
}

```

```

pool.allowCoreThreadTimeOut(true);
System.out.println(pool.allowsCoreThreadTimeOut());
for (int i = 0; i < 4; i++) {
    MyRunnable runnable = new MyRunnable();
    pool.execute(runnable);
}
Thread.sleep(8000);
System.out.println(pool.getPoolSize());
}
}

```

程序运行结果如图 4-45 所示。

```

false
pool-1-thread-1 begin 1434157918421
pool-1-thread-3 begin 1434157918421
pool-1-thread-4 begin 1434157918421
pool-1-thread-2 begin 1434157918421
pool-1-thread-4 end 1434157918421
pool-1-thread-3 end 1434157918421
pool-1-thread-1 end 1434157918421
pool-1-thread-2 end 1434157918421
4

```

图 4-44 运行结果

```

true
pool-1-thread-1 begin 1434157980609
pool-1-thread-3 begin 1434157980609
pool-1-thread-2 begin 1434157980609
pool-1-thread-3 end 1434157980609
pool-1-thread-1 end 1434157980609
pool-1-thread-4 begin 1434157980609
pool-1-thread-2 end 1434157980609
pool-1-thread-4 end 1434157980609
0

```

图 4-45 运行结果

4.3.9 方法 prestartCoreThread() 和 prestartAllCoreThreads()

方法 prestartCoreThread() 每调用一次就创建一个核心线程，返回值为 boolean，含义是是否启动了。

方法 prestartAllCoreThreads() 的作用是启动全部核心线程，返回值是启动核心线程的数量。创建测试用的项目 ThreadPoolExecutor_9，类 Run1.java 代码如下：

```

package test.run;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run1 {
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println("打印了! begin "
                        + Thread.currentThread().getName());
                    Thread.sleep(4000);
                    System.out.println("打印了! end "
                        + Thread.currentThread().getName());
                } catch (InterruptedException e) {

```

```

// TODO Auto-generated catch block
e.printStackTrace();
}
}

ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 2, 5,
    TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
System.out.println("线程池中的线程数 A: " + executor.getPoolSize());
System.out.println("Z1=" + executor.prestartCoreThread());
System.out.println("线程池中的线程数 B: " + executor.getPoolSize());
System.out.println("Z2=" + executor.prestartCoreThread());
System.out.println("线程池中的线程数 C: " + executor.getPoolSize());
System.out.println("Z3=" + executor.prestartCoreThread()); // 无效代码
System.out.println("Z4=" + executor.prestartCoreThread()); // 无效代码
System.out.println("Z5=" + executor.prestartCoreThread()); // 无效代码
System.out.println("Z6=" + executor.prestartCoreThread()); // 无效代码
System.out.println("线程池中的线程数 D: " + executor.getPoolSize());
}
}

```

程序运行结果如图 4-46 所示。

创建类 Run2.java 代码如下：

```

package test.run;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run2 {
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("打印了! begin "
                    + Thread.currentThread().getName());
                System.out.println("打印了! end "
                    + Thread.currentThread().getName());
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 2, 5,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        System.out.println("线程池中的线程数 A: " + executor.getPoolSize());
        System.out.println("启动核心线程数量为: " +
            executor.prestartAllCoreThreads());
        System.out.println("线程池中的线程数 B: " + executor.getPoolSize());
    }
}

```

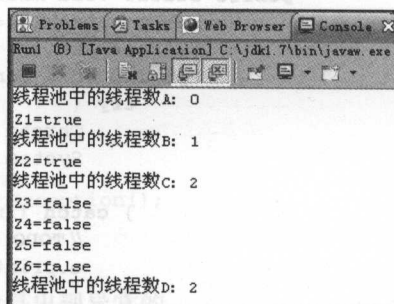


图 4-46 运行结果

程序运行结果如图 4-47 所示。

4.3.10 方法 getCompletedTaskCount()

方法 getCompletedTaskCount() 的作用是取得已经执行完成的任务数。

创建测试用的项目 ThreadPoolExecutor_10，类 Run1。

java 代码如下：

```
package test.run;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run1 {
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                    System.out.println("打印了！"
                        + Thread.currentThread().getName());
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 2, 5,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        executor.execute(runnable);
        executor.execute(runnable);
        executor.execute(runnable);
        executor.execute(runnable);
        executor.execute(runnable);
        executor.execute(runnable);
        Thread.sleep(1000);
        System.out.println(executor.getCompletedTaskCount());
        Thread.sleep(1000);
        System.out.println(executor.getCompletedTaskCount());
        Thread.sleep(1000);
        System.out.println(executor.getCompletedTaskCount());
        Thread.sleep(1000);
        System.out.println(executor.getCompletedTaskCount());
        Thread.sleep(1000);
        System.out.println(executor.getCompletedTaskCount());
    }
}
```

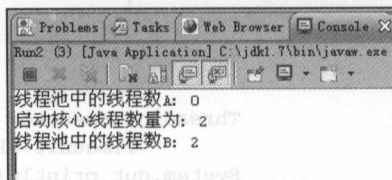


图 4-47 运行结果

程序运行结果如图 4-48 所示。

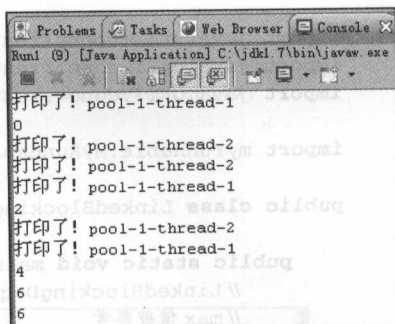
4.3.11 常见 3 种队列结合 max 值的因果效果

在使用 ThreadPoolExecutor 线程池的过程中，常使用 3 种队列 ArrayBlockingQueue、LinkedBlockingDeque 和 SynchronousQueue。

其中 ArrayBlockingQueue 和 LinkedBlockingDeque 类可以指定队列存储元素的多少，ArrayBlockingQueue 类构造方法代码如下：

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull = lock.newCondition();
}
```

图 4-48 运行结果



LinkedBlockingDeque 类构造方法代码如下：

```
public LinkedBlockingDeque(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
}
```

从源代码中可以发现，如果传入 capacity 值是 ≤ 0 时是出现异常的。

那么本节要实验的环境是欲执行任务的数量大于 ThreadPoolExecutor 线程池的 maximum-PoolSize 最大值时，在不同队列中会出现什么样的情况。

创建项目 queueDiff_big_test，创建 MyRunnable.java 类代码如下：

```
package myrunnable;

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        try {
            System.out.println("begin " + System.currentTimeMillis());
            Thread.sleep(1000);
            System.out.println(" end " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

创建 LinkedBlockingDequeTest1.java 类代码如下：

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class LinkedBlockingDequeTest1 {

    public static void main(String[] args) throws InterruptedException {
        // LinkedBlockingDeque 使用带参构造
        // max 值被参考
        LinkedBlockingDeque linked = new LinkedBlockingDeque<Runnable>(2);
        System.out.println(linked.size());
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 5,
            TimeUnit.SECONDS, linked);
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        System.out.println(pool.getPoolSize() + " " + linked.size());
        // 放入队列 2 个任务，执行 3 个任务
    }
}

```

运行结果如图 4-49 所示。

从运行结果得知，队列中存放的元素个数为 2，正在执行的线程任务数为 3。

创建 LinkedBlockingDequeTest2.java 类代码如下：

```

package test;

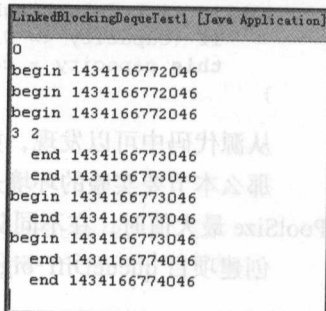
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class LinkedBlockingDequeTest2 {

    public static void main(String[] args) throws InterruptedException {
        // LinkedBlockingDeque 使用带参构造
        // max 值被参考
        // 但队列容量不够，有一个任务出现异常
        LinkedBlockingDeque linked = new LinkedBlockingDeque<Runnable>(2);
        System.out.println(linked.size());
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 5,
            TimeUnit.SECONDS, linked);
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
    }
}

```



```

LinkedBlockingDequeTest1 [Java Application]
0
begin 1434166772046
begin 1434166772046
begin 1434166772046
3 2
end 1434166773046
end 1434166773046
begin 1434166773046
end 1434166773046
begin 1434166773046
end 1434166774046
end 1434166774046

```

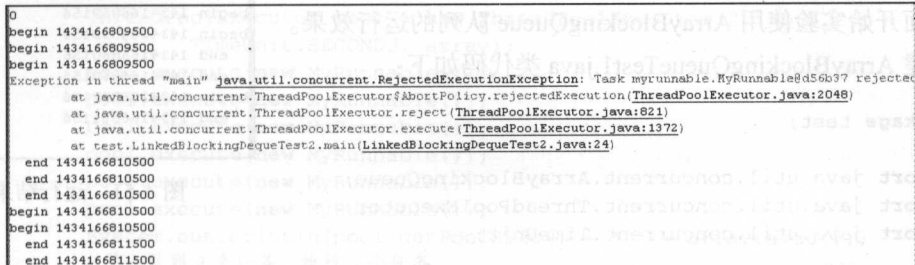
图 4-49 运行结果

```

pool.execute(new MyRunnable());
pool.execute(new MyRunnable());
pool.execute(new MyRunnable());
pool.execute(new MyRunnable());
System.out.println(pool.getPoolSize() + " " + linked.size());
// 放入队列 2 个任务, 执行 3 个任务
// 有 1 个任务被拒绝
}
}

```

运行结果如图 4-50 所示。



```

0
begin 1434166809500
begin 1434166809500
begin 1434166809500
Exception in thread "main" java.util.concurrent.RejectedExecutionException: Task MyRunnable.MyRunnableId5637 rejected
at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:2048)
at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:821)
at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1372)
at test.LinkedBlockingDequeTest2.main(LinkedBlockingDequeTest2.java:24)
end 1434166810500
end 1434166810500
end 1434166810500
begin 1434166810500
begin 1434166810500
end 1434166811500
end 1434166811500

```

图 4-50 运行结果

从运行结果可知, 其中一个任务被拒绝执行, 其他任务正常执行。

创建 `LinkedBlockingDequeTest3.java` 类代码如下:

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class LinkedBlockingDequeTest3 {

    public static void main(String[] args) throws InterruptedException {
        // LinkedBlockingDeque 使用无参构造
        // max 值被忽略
        LinkedBlockingDeque linked = new LinkedBlockingDeque<Runnable>();
        System.out.println(linked.size());
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 5,
            TimeUnit.SECONDS, linked);
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        System.out.println(pool.getPoolSize() + " " + linked.size());
        // 使用 new LinkedBlockingDeque<Runnable>(); 无参构造
        // capacity 值是 Integer.MAX_VALUE
    }
}

```

```

// 源代码如下:
// public LinkedBlockingDeque() {
// this(Integer.MAX_VALUE);
// }
// 说明 LinkedBlockingDeque 队列
// 里面可以存储 Integer.MAX_VALUE
// 个数数据
// 放入队列 3 个任务, 执行 2 个任务
}
}

```

运行结果如图 4-51 所示。

下面开始实验使用 ArrayBlockingQueue 队列的运行效果。

创建 ArrayBlockingQueueTest1.java 类代码如下:

```

package test;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class ArrayBlockingQueueTest1 {

    public static void main(String[] args) throws InterruptedException {
        // ArrayBlockingQueue 使用带参构造
        // max 值被参考
        ArrayBlockingQueue array = new ArrayBlockingQueue(2);
        System.out.println(array.size());
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 5,
            TimeUnit.SECONDS, array);
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        System.out.println(pool.getPoolSize() + " " + array.size());
        // 放入队列 2 个任务, 执行 3 个任务
    }
}

```

运行结果如图 4-52 所示。

队列中存储元素的个数为 2, 正在运行的线程数为 3。

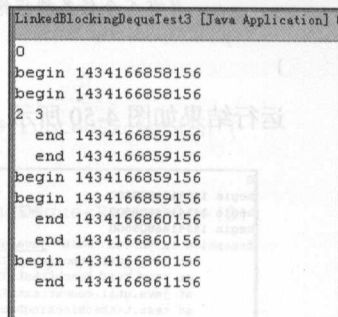
创建 ArrayBlockingQueueTest2.java 类代码如下:

```

package test;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;

```

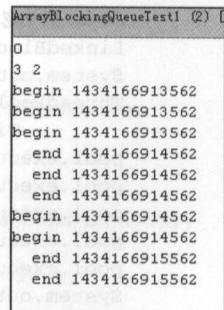


```

LinkedBlockingDequeTest3 [Java Application]
0
begin 1434166858156
begin 1434166858156
2 3
end 1434166859156
end 1434166859156
begin 1434166859156
begin 1434166859156
end 1434166860156
end 1434166860156
begin 1434166860156
end 1434166861156

```

图 4-51 运行结果



```

ArrayBlockingQueueTest1 (2)
0
3 2
begin 1434166913562
begin 1434166913562
begin 1434166913562
end 1434166914562
end 1434166914562
end 1434166914562
begin 1434166914562
begin 1434166914562
end 1434166915562
end 1434166915562

```

图 4-52 运行结果

```

import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class ArrayBlockingQueueTest2 {

    public static void main(String[] args) throws InterruptedException {
        // ArrayBlockingQueue 使用带参构造
        // max 值被参考
        // 但队列容量不够, 有一个任务出现异常
        ArrayBlockingQueue array = new ArrayBlockingQueue(2);
        System.out.println(array.size());
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 5,
            TimeUnit.SECONDS, array);
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        System.out.println(pool.getPoolSize() + " " + array.size());
        // 放入队列 2 个任务, 执行 3 个任务
        // 有 1 个任务被拒绝
    }
}

```

运行结果如图 4-53 所示。

```

0
begin 1434166953750
begin 1434166953750
begin 1434166953750
Exception in thread "main" java.util.concurrent.RejectedExecutionException: Task myrunnable.MyRunnable@51b2c rejected
at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:2048)
at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:821)
at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1372)
at test.ArrayBlockingQueueTest2.main(ArrayBlockingQueueTest2.java:24)
end 1434166954750
end 1434166954750
end 1434166954750
begin 1434166954750
begin 1434166954750
end 1434166955750
end 1434166955750

```

图 4-53 运行结果

创建 SynchronousQueueTest1.java 类代码如下：

```

package test;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class SynchronousQueueTest1 {

```



```

public static void main(String[] args) throws InterruptedException {
    SynchronousQueue linked = new SynchronousQueue<Runnable>();
    System.out.println(linked.size());
    ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 5,
        TimeUnit.SECONDS, linked);
    pool.execute(new MyRunnable());
    pool.execute(new MyRunnable());
    pool.execute(new MyRunnable());
    System.out.println(pool.getPoolSize() + " " + linked.size());
    // 直接执行 3 个任务
}
}

```

运行结果如图 4-54 所示。

创建 SynchronousQueueTest2.java 类代码如下：

```

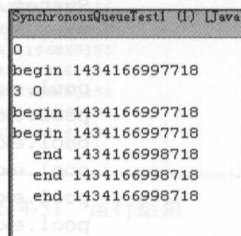
package test;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class SynchronousQueueTest2 {

```



```

SynchronousQueueTest1 () [Java]
0
begin 1434166997718
3 0
begin 1434166997718
begin 1434166997718
end 1434166998718
end 1434166998718
end 1434166998718

```

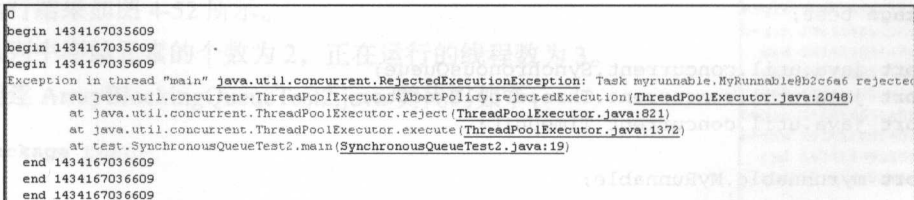
图 4-54 运行结果

```

    public static void main(String[] args) throws InterruptedException {
        SynchronousQueue linked = new SynchronousQueue<Runnable>();
        System.out.println(linked.size());
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 5,
            TimeUnit.SECONDS, linked);
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        pool.execute(new MyRunnable());
        System.out.println(pool.getPoolSize() + " " + linked.size());
        // 直接执行 3 个任务
        // 有一个任务被拒绝
    }
}

```

运行结果如图 4-55 所示。



```

0
begin 1434167035609
begin 1434167035609
begin 1434167035609
Exception in thread "main" java.util.concurrent.RejectedExecutionException: Task myrunnable.MyRunnable@b2c6a6 rejected
at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:2048)
at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:821)
at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1372)
at test.SynchronousQueueTest2.main(SynchronousQueueTest2.java:19)
end 1434167036609
end 1434167036609
end 1434167036609

```

图 4-55 运行结果

从运行结果可知,有3个任务正常执行,另外一个任务被拒绝执行,因为线程数已经超过 max 值。

4.3.12 线程池 ThreadPoolExecutor 的拒绝策略

线程池中的资源全部被占用的时候,对新添加的 Task 任务有不同的处理策略,在默认的情况下,ThreadPoolExecutor 类中有4种不同的处理方式:

- ❑ AbortPolicy: 当任务添加到线程池中被拒绝时,它将抛出 RejectedExecutionException 异常。
- ❑ CallerRunsPolicy: 当任务添加到线程池中被拒绝时,会使用调用线程池的 Thread 线程对象处理被拒绝的任务。
- ❑ DiscardOldestPolicy: 当任务添加到线程池中被拒绝时,线程池会放弃等待队列中最旧的未处理任务,然后将被拒绝的任务添加到等待队列中。
- ❑ DiscardPolicy: 当任务添加到线程池中被拒绝时,线程池将丢弃被拒绝的任务。

1. AbortPolicy 策略

AbortPolicy 策略是当任务添加到线程池中被拒绝时,它将抛出 RejectedExecutionException 异常。

创建实验用的项目 Policy_AbortPolicy, 创建类 Run.java 代码如下:

```
package test.run;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run {

    public static void main(String[] args) {

        Runnable runnable = new Runnable() {

            public void run() {

                try {

                    Thread.sleep(5000);

                    System.out.println(Thread.currentThread().getName()

                        + " run end!");

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 3, 5,

            TimeUnit.SECONDS, new ArrayBlockingQueue(2),

            new ThreadPoolExecutor.AbortPolicy());

        executor.execute(runnable); // 不报错

        executor.execute(runnable); // 不报错
```

```

        executor.execute(runnable);    // 不报错
        executor.execute(runnable);    // 不报错
        executor.execute(runnable);    // 不报错
        // executor.execute(runnable); // 报错
    }
}

```

程序运行后不出现异常，如图 4-56 所示。

将代码：

```
// executor.execute(runnable); // 报错
```

注释去掉后，出现异常，超出线程池容量，如图 4-57 所示。

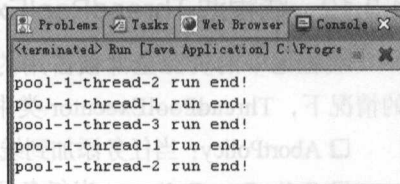


图 4-56 不出现异常

```

Exception in thread "main" java.util.concurrent.RejectedExecutionException
    at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:17)
    at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:767)
    at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:658)
    at test.Run.main(Run.java:31)
pool-1-thread-3 run end!
pool-1-thread-1 run end!
pool-1-thread-2 run end!
pool-1-thread-3 run end!
pool-1-thread-1 run end!

```

图 4-57 出现异常

使用 AbortPolicy 策略后，线程数量超出 max 值时，线程池抛出 java.util.concurrent.RejectedExecutionException 异常。

2. CallerRunsPolicy 策略

CallerRunsPolicy 策略是当任务添加到线程池中被拒绝时，会使用调用线程池的 Thread 线程对象处理被拒绝的任务。

创建实验用的项目 Policy_CallerRunsPolicy_1，线程类 MyThreadA.java 代码如下：

```

package extthread;

public class MyThreadA extends Thread {

    @Override
    public void run() {
        try {
            Thread.sleep(5000);
            System.out.println("  end " + Thread.currentThread().getName()
                + " " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

创建类 Run.java 代码如下：

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import extthread.MyThreadA;

public class Run {

    public static void main(String[] args){
        MyThreadA a = new MyThreadA();
        LinkedBlockingDeque queue = new LinkedBlockingDeque(2);
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 5,
            TimeUnit.SECONDS, queue,
            new ThreadPoolExecutor.CallerRunsPolicy());
        System.out.println("a begin " + Thread.currentThread().getName() + " "
            + System.currentTimeMillis());
        pool.execute(a);
        pool.execute(a);
        pool.execute(a);
        pool.execute(a);
        pool.execute(a);
        pool.execute(a);
        System.out.println("a end " + Thread.currentThread().getName() + " "
            + System.currentTimeMillis());
    }
}

```

程序运行结果如图 4-58 所示。

在上面的实验中，线程 main 被阻塞，严重影响程序的运行效率，所以并不建议这样做，通过改变代码结构可以改善这种情况，创建名称为 Policy_CallerRunsPolicy_2 的项目，类 MyThreadA.java 代码如下：

```

package extthread;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class MyThreadA extends Thread {

    @Override
    public void run() {
        MyThreadB b = new MyThreadB();

        LinkedBlockingDeque queue = new LinkedBlockingDeque(2);
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 5,

```

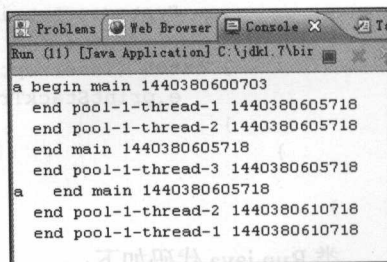


图 4-58 由 main 线程执行的任务

```

        TimeUnit.SECONDS, queue,
        new ThreadPoolExecutor.CallerRunsPolicy());
    System.out.println("a begin " + Thread.currentThread().getName() + " "
        + System.currentTimeMillis());
    pool.execute(b);
    pool.execute(b);
    pool.execute(b);
    pool.execute(b);
    pool.execute(b);
    pool.execute(b);
    System.out.println("a end " + Thread.currentThread().getName() + " "
        + System.currentTimeMillis());
    System.out.println("a end " + System.currentTimeMillis());
}
}

```

类 MyThreadB.java 代码如下:

```

package extthread;

public class MyThreadB extends Thread {

    @Override
    public void run() {
        try {
            Thread.sleep(5000);
            System.out.println(" end " + Thread.currentThread().getName()
                + " " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

类 Run.java 代码如下:

```

package test;

import extthread.MyThreadA;

public class Run {

    public static void main(String[] args) {
        MyThreadA a = new MyThreadA();
        a.setName("AA");
        a.start();
        System.out.println("main end!");
    }
}

```

程序运行结果如图 4-59 所示。

3. DiscardOldestPolicy 策略

DiscardOldestPolicy 策略是当任务添加到线程池中被拒绝时，线程池会放弃等待队列中最旧的未处理任务，然后将被拒绝的任务添加到等待队列中。

创建实验用的项目 Policy_DiscardOldestPolicy，创建类 MyRunnable.java 代码如下：

```
package test.run;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void run() {
        try {
            System.out.println(username + " run");
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

创建类 Run.java 代码如下：

```
package test.run;

import java.util.Iterator;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        // 创建线程池
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            10, 10, 1, TimeUnit.SECONDS,
            new ArrayBlockingQueue<Runnable>(10),
            new DiscardOldestPolicy(),
            r -> {
                // 线程池满了，拒绝任务
            }
        );

        // 提交任务
        executor.execute(new MyRunnable("a"));
        executor.execute(new MyRunnable("a"));
        executor.execute(new MyRunnable("a"));
        executor.execute(new MyRunnable("a"));
        executor.execute(new MyRunnable("a"));
        executor.execute(new MyRunnable("a"));
        executor.execute(new MyRunnable("a"));
        executor.execute(new MyRunnable("a"));
        executor.execute(new MyRunnable("a"));
        executor.execute(new MyRunnable("a"));

        // 等待线程池执行完毕
        executor.awaitTermination(10, TimeUnit.SECONDS);
    }
}
```

```
main end!
a begin AA 1440380793187
end pool-1-thread-1 1440380798187
end pool-1-thread-2 1440380798187
end pool-1-thread-3 1440380798187
end AA 1440380798187
a end AA 1440380798187
a end 1440380798187
end pool-1-thread-2 1440380803187
end pool-1-thread-1 1440380803187
```

图 4-59 线程 main 并未被阻塞


```

public static void main(String[] args) throws InterruptedException {
    ArrayBlockingQueue queue = new ArrayBlockingQueue(2);
    ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 3, 5,
        TimeUnit.SECONDS, queue,
        new ThreadPoolExecutor.DiscardOldestPolicy());
    for (int i = 0; i < 5; i++) {
        MyRunnable runnable = new MyRunnable("Runnable" + (i + 1));
        executor.execute(runnable);
    }
    Thread.sleep(50);
    Iterator iterator = queue.iterator();
    while (iterator.hasNext()) {
        Object object = iterator.next();
        System.out.println(((MyRunnable) object).getUsername());
    }
    executor.execute(new MyRunnable("Runnable6"));
    executor.execute(new MyRunnable("Runnable7"));
    iterator = queue.iterator();
    while (iterator.hasNext()) {
        Object object = iterator.next();
        System.out.println(((MyRunnable) object).getUsername());
    }
}

```

程序运行结果如图 4-60 所示。

4. DiscardPolicy 策略

DiscardPolicy 策略是当任务添加到线程池中被拒绝时，线程池将丢弃被拒绝的任务。

创建实验用的项目 Policy_DiscardPolicy，创建类 Run.java 代码如下：

```

package test.run;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Run {

    public static void main(String[] args) throws InterruptedException {

        Runnable runnable = new Runnable() {
            public void run() {
                try {
                    Thread.sleep(5000);
                    System.out.println(Thread.currentThread().getName()
                        + " run end!");
                }
            }
        };
    }
}

```

```

Terminated Run (2) [J
Runnable2 run
Runnable1 run
Runnable5 run
Runnable3
Runnable4
Runnable6
Runnable7
Runnable6 run
Runnable7 run

```

图 4-60 早期的任务 3 和任务 4 被取消

```

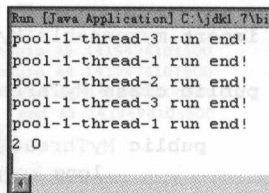
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

};

ArrayBlockingQueue queue = new ArrayBlockingQueue(2);
ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 3, 5,
    TimeUnit.SECONDS, queue, new ThreadPoolExecutor.DiscardPolicy());
executor.execute(runnable);
executor.execute(runnable);
executor.execute(runnable);
executor.execute(runnable);
executor.execute(runnable);
executor.execute(runnable);
executor.execute(runnable);
executor.execute(runnable);
Thread.sleep(8000);
System.out.println(executor.getPoolSize() + " " + queue.size());
}
}

```

程序运行结果如图 4-61 所示。



```

Run [Java Application] C:\jdk1.7\bin
pool-1-thread-3 run end!
pool-1-thread-1 run end!
pool-1-thread-2 run end!
pool-1-thread-3 run end!
pool-1-thread-1 run end!
2 0

```

图 4-61 多余的任务被取消执行

4.3.13 方法 afterExecute() 和 beforeExecute()

在线程池 ThreadPoolExecutor 类中重写这两个方法可以对线程池中执行的线程对象实现监控。

创建实验用的项目 ThreadPoolExecutor_after_before, 类 MyRunnable.java 代码如下:

```

package myrunnable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public void run() {

```

```

try {
    System.out.println(" 打印了! begin " + username + " "
        + System.currentTimeMillis());
    Thread.sleep(4000);
    System.out.println(" 打印了!          end " + username + " "
        + System.currentTimeMillis());
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

```

类 `MyThreadPoolExecutor.java` 代码如下:

```

package executor;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class MyThreadPoolExecutor extends ThreadPoolExecutor {

    public MyThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
        long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
    }

    @Override
    protected void afterExecute(Runnable r, Throwable t) {
        super.afterExecute(r, t);
        System.out.println(((MyRunnable) r).getUsername() + " 执行完了");
    }

    @Override
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        System.out.println(" 准备执行: " + ((MyRunnable) r).getUsername());
    }
}

```

类 `Run.java` 代码如下:

```

package test.run;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;
import executor.MyThreadPoolExecutor;

```

```

public class Run {

    public static void main(String[] args) throws InterruptedException {

        MyThreadPoolExecutor executor = new MyThreadPoolExecutor(2, 2,
            Integer.MAX_VALUE, TimeUnit.SECONDS,
            new LinkedBlockingDeque<Runnable>());
        executor.execute(new MyRunnable("A1"));
        executor.execute(new MyRunnable("A2"));
        executor.execute(new MyRunnable("A3"));
        executor.execute(new MyRunnable("A4"));
    }
}

```

程序运行结果如图 4-62 所示。

4.3.14 方法 remove(Runnable) 的使用

方法 remove(Runnable) 可以删除尚未被执行的 Runnable 任务。

创建实验用的项目 ThreadPoolExecutor_remove, 类 Test1.java 代码如下:

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Test1 {

    public static void main(String[] args) throws InterruptedException {

        Runnable runnable1 = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName()
                        + " begin");
                    Thread.sleep(5000);
                    System.out.println(Thread.currentThread().getName()
                        + " end");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 100,
            TimeUnit.SECONDS, new LinkedBlockingDeque());
        executor.execute(runnable1);
    }
}

```

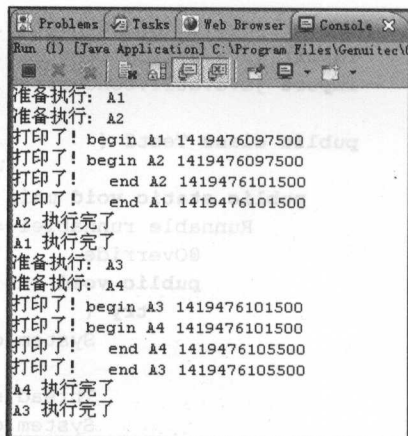


图 4-62 运行结果抓取状态

```

        Thread.sleep(1000);
        executor.remove(runnable1);
        System.out.println("任务正在运行不能删除");
    }
}

```

程序运行结果如图 4-63 所示。

继续实验，创建类 Test2.java 代码如下：

```

pool-1-thread-1 begin
任务正在运行不能删除
pool-1-thread-1 end

```

图 4-63 运行结果

```
package test;
```

```

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Test2 {

    public static void main(String[] args) throws InterruptedException {
        Runnable runnable1 = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName()
                        + " begin");
                    Thread.sleep(5000);
                    System.out.println(Thread.currentThread().getName()
                        + " end");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        Runnable runnable2 = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName()
                        + " begin");
                    Thread.sleep(5000);
                    System.out.println(Thread.currentThread().getName()
                        + " end");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 100,
            TimeUnit.SECONDS, new LinkedBlockingDeque());
        executor.execute(runnable1);
        executor.execute(runnable2);
    }
}

```

Import

The

程序运行结果如图 4-64 所示。

图 4-64 运行结果

remove() 方法却不能删除此任务。

创建 Test3.java，代码如下：

```
package test;
```

```
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Test3 {
```

```
public static void main(String[] args) {
```

```
try {
```

```
Runnable runnable1 = new Runnable() {
```

```
@Override
```

```
public void run() {
```

```
try {
```

```
System.out.println("beginA "
```

```
+ Thread.currentThread().getName() + " "
```

```
+ System.currentTimeMillis());
```

```
Thread.sleep(5000);
```

```
System.out.println("  endA  ")
```

```
+ Thread.currentThread().getName() + " "
```

```
+ System.currentTimeMillis());
```

```
} catch (InterruptedException e) {
```

```
e.printStackTrace();
```

}

} ;

```
Runnable runnable2 = new Runnable() {
```

```
@Override
```

```
public void run() {
```

```
try {
```

```
System.out.println("beginB "
```

```
+ Thread.currentThread().getName() + " "
```

```
+ System.currentTimeMillis());
```

```
Thread.sleep(5000);
```

```
System.out.println("  endB  ")
```

```
+ Thread.currentThread().getName() + " "
```

```
+ System.currentTimeMillis());
```

```
} catch (InterruptedException e) {
```



```

        e.printStackTrace();
    }
}

ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 5,
    TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
executor.submit(runnable1);
executor.submit(runnable2);
Thread.sleep(1000);
executor.remove(runnable2);
System.out.println("main end!");
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

```

运行结果如图 4-65 所示。

任务 2 没有被删除。

```

beginA pool-1-thread-1 1434416105015
main end!
endA pool-1-thread-1 1434416110015
beginB pool-1-thread-1 1434416110015
endB pool-1-thread-1 1434416115015

```

图 4-65 运行结果

4.3.15 多个 get 方法的测试

线程池 `ThreadPoolExecutor` 有很多 `getX()` 方法，熟悉这些方法是观察线程池状态最好的方式。

创建测试用的项目 `get_diff`。

(1) 方法 `getActiveCount()` 的测试

方法 `getActiveCount()` 的作用是取得有多少个线程正在执行任务。

线程类 `MyThreadA.java` 代码如下：

```

package extthread;

public class MyThreadA extends Thread {

    @Override
    public void run() {
        try {
            System.out.println(" begin " + Thread.currentThread().getName()
                + " " + System.currentTimeMillis() + " 运行中");
            Thread.sleep(5000);
            System.out.println(" end " + Thread.currentThread().getName()
                + " " + System.currentTimeMillis() + " 运行中");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

类 `getActiveCount_test1.java` 代码如下:

```
package test;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import extthread.MyThreadA;

public class getActiveCount_test1 {

    public static void main(String[] args) throws InterruptedException
    {
        try {
            MyThreadA a = new MyThreadA();

            SynchronousQueue queue = new SynchronousQueue();
            ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 5, 5,
                TimeUnit.SECONDS, queue);
            pool.execute(a);
            pool.execute(a);
            pool.execute(a);
            System.out
                .println(pool.getActiveCount() + " " + pool.getPoolSize());
            Thread.sleep(7000);
            System.out
                .println(pool.getActiveCount() + " " + pool.getPoolSize());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 4-66 所示。

方法 `getPoolSize()` 获得的是当前线程池里面有多少个线程, 这些线程数包括正在执行任务的线程, 也包括正在休眠的线程。

方法 `getActiveCount()` 获得正在执行任务的线程数。

(2) 方法 `getCompletedTaskCount()` 的测试

方法 `getCompletedTaskCount()` 的作用是取得有多少个线程已经执行完任务了。

类 `getCompletedTaskCount_test1.java` 代码如下:

```
package test;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class getCompletedTaskCount_test1 {
```

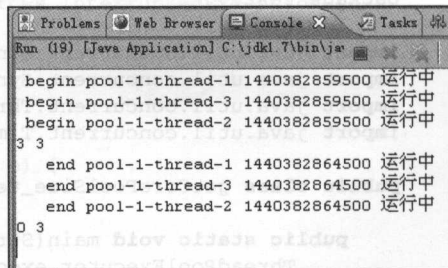


图 4-66 运行结果



```

public static void main(String[] args) throws InterruptedException {
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    };

    ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 5, 100,
        TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
    executor.execute(runnable);
    executor.execute(runnable);
    executor.execute(runnable);
    System.out.println(executor.getCompletedTaskCount());
    Thread.sleep(7000);
    System.out.println(executor.getCompletedTaskCount());
}

```

getComplete
0
3

程序运行结果如图 4-67 所示。

图 4-67 运行结果

(3) 方法 `getCorePoolSize()` 的测试

方法 `getCorePoolSize()` 的作用是取得构造方法传入的 `corePoolSize` 参数值。

类 `getCorePoolSize_test1.java` 代码如下：

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class getCorePoolSize_test1 {

    public static void main(String[] args) throws InterruptedException {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 5, 100,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
        System.out.println("getCorePoolSize=" +
            executor.getCorePoolSize());
    }
}

```

getCorePoolSize=2

程序运行结果如图 4-68 所示。

图 4-68 运行结果

(4) 方法 `getMaximumPoolSize()` 的测试

方法 `getMaximumPoolSize()` 的作用是取得构造方法传入的 `maximumPoolSize` 参数值。

类 `getMaximumPoolSize_test1.java` 代码如下：

```

package test;

```

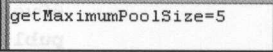
```

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class getMaximumPoolSize_test1 {

    public static void main(String[] args) throws InterruptedException {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 5, 100,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
        System.out.println("getMaximumPoolSize=" +
            executor.getMaximumPoolSize());
    }
}

```



程序运行结果如图 4-69 所示。

图 4-69 运行结果

(5) 方法 getPoolSize() 的测试

方法 getPoolSize() 的作用是取得池中有多少个线程。

类 getPoolSize_test1.java 代码如下：

```

package test;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

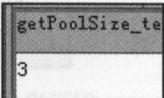
public class getPoolSize_test1 {

    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 5, 100,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
        executor.execute(runnable);
        executor.execute(runnable);
        executor.execute(runnable);
        System.out.println(executor.getPoolSize());
    }
}

```

程序运行结果如图 4-70 所示。



(6) 方法 getTaskCount() 的测试

方法 getTaskCount() 的作用是取得有多少个任务发送给了线程池。

图 4-70 运行结果

类 `getTaskCount_test1.java` 代码如下：

```
package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class getTaskCount_test1 {

    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 5, 100,
            TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
        for (int i = 0; i < 10; i++) {
            executor.execute(runnable);
        }
        System.out.println(executor.getTaskCount());
    }
}
```

getTaskCount
10

程序运行结果如图 4-71 所示。

图 4-71 运行结果

4.3.16 线程池 `ThreadPoolExecutor` 与 `Runnable` 执行为乱序特性

接口 `Runnable` 在 `ThreadPoolExecutor` 的队列中是按顺序取出，执行却是乱序的。

创建测试用的项目 `runnable_asynchronized_test`，类 `MyRunnable.java` 代码如下：

```
package myrunnable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    @Override
    public void run() {
```

```

        System.out.println(username);
    }
}

```

类 Test.java 代码如下:

```

package test;

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable;

public class Test {

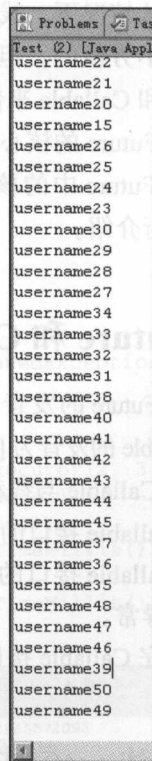
    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 10,
            TimeUnit.SECONDS, new LinkedBlockingDeque());
        for (int i = 0; i < 50; i++) {
            MyRunnable myrunnable = new MyRunnable("username" + (i + 1));
            executor.execute(myrunnable);
        }
    }
}

```

程序运行结果如图 4-72 所示。

4.4 本章总结

本章主要介绍 ThreadPoolExecutor 类的构造方法中各个参数的作用与使用效果,还介绍了 Executors 工厂类常用 API 的使用,也将大部分 ThreadPoolExecutor 线程池类的常见 API 一同进行了介绍,并且对 ThreadPoolExecutor 线程池的拒绝策略进行了实验,通过使用线程池能最大程度地减少创建线程对象的内存与 CPU 开销,加快程序运行效率,也对创建线程类的代码进行了封装,方便开发并发类型的软件项目。



```

Test (2) [Java Appl
username22
username21
username20
username15
username26
username25
username24
username23
username30
username29
username28
username27
username34
username33
username32
username31
username38
username40
username41
username42
username43
username44
username45
username37
username36
username35
username48
username47
username46
username39
username50
username49

```

图 4-72 乱序打印

Future 和 Callable 的使用

在默认情况下，线程 Thread 对象不具有返回值的功能，如果在需要取得返回值的情况下是极为不方便的，但在 Java1.5 的并发包中可以使用 Future 和 Callable 来使线程具有返回值的功能。

接口 Future 的核心方法如图 5-1 所示。

接口 Future 中的核心方法都会在本章中以案例的方式进行介绍。

```

cancel(boolean mayInterruptIfRunning) : boolean - Future
equals(Object obj) : boolean - Object
get() : Object - Future
get(long timeout, TimeUnit unit) : Object - Future
getClass() : Class<?> - Object
hashCode() : int - Object
isCancelled() : boolean - Future
isDone() : boolean - Future
notify() : void - Object
notifyAll() : void - Object
toString() : String - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object

```

图 5-1 接口 Future 的核心方法

5.1 Future 和 Callable 的介绍

单词 Future 的发音为 [ˈfju:tʃə]，中文翻译是将来、未来。Callable 的发音为 [ˈkɔ:ləbl]，中文翻译为可随时支取的，请求即付的，随时处理/可偿还的。接口 Callable 与线程功能密不可分，但和 Runnable 的主要区别为：

- 1) Callable 接口的 call() 方法可以有返回值，而 Runnable 接口的 run() 方法没有返回值。
- 2) Callable 接口的 call() 方法可以声明抛出异常，而 Runnable 接口的 run() 方法不可以声明抛出异常。

执行完 Callable 接口中的任务后，返回值是通过 Future 接口进行获得的。

5.2 方法 get() 结合 ExecutorService 中的 submit(Callable<T>) 的使用

方法 submit(Callable<T>) 可以执行参数为 Callable 的任务。

方法 `get()` 用于获得返回值。

创建实验用的项目 `future_callable_1`，创建类 `MyCallable.java` 代码如下：

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    private int age;

    public MyCallable(int age) {
        super();
        this.age = age;
    }

    public String call() throws Exception {
        Thread.sleep(8000);
        return "返回值 年龄是: " + age;
    }
}
```

创建类 `Run.java` 代码如下：

```
package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallable;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        try {
            MyCallable callable = new MyCallable(100);

            ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 3, 5L,
                TimeUnit.SECONDS, new LinkedBlockingDeque());
            Future<String> future = executor.submit(callable);
            System.out.println("main A " + System.currentTimeMillis());
            System.out.println(future.get());
            System.out.println("main B " + System.currentTimeMillis());
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 5-2 所示。

从控制台打印的结果来看，方法 `get()` 具有阻塞特性。

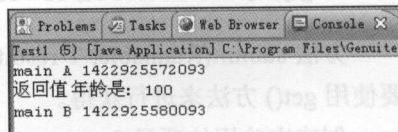


图 5-2 运行结果

5.3 方法 get() 结合 ExecutorService 中的 submit(Runnable) 和 isDone() 的使用

方法 submit() 不仅可以传入 Callable 对象，也可以传入 Runnable 对象，说明 submit() 方法支持有返回值和无返回值的功能。

创建实验用的项目 future_callable_2，创建类 Run.java 代码如下：

```
package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Run {

    public static void main(String[] args) {
        try {
            Runnable runnable = new Runnable() {
                @Override
                public void run() {
                    System.out.println("打印的信息");
                }
            };
            ExecutorService executorRef = Executors.newCachedThreadPool();
            Future future = executorRef.submit(runnable);
            System.out.println(future.get() + " " + future.isDone());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 5-3 所示。

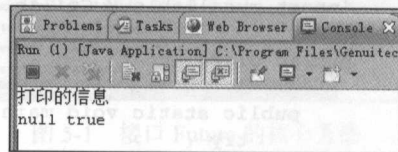


图 5-3 运行结果

通过此实验可得知，如果 submit() 方法传入 Callable 接口则可以有返回值，如果传入 Runnable 则无返回值，打印的结果就是 null。方法 get() 具有阻塞特性，而 isDone() 方法无阻塞特性。

5.4 使用 ExecutorService 接口中的方法 submit(Runnable, T result)

方法 submit(Runnable, T result) 的第 2 个参数 result 可以作为执行结果的返回值，而不需要使用 get() 方法来进行获得。

创建实验用的项目 future_callable_3，实体类 Userinfo.java 代码如下：

```

package entity;

public class Userinfo {

    private String username;
    private String password;

    public Userinfo() {
        super();
    }

    public Userinfo(String username, String password) {
        super();
        this.username = username;
        this.password = password;
    }

    // 其他 set 及 get 方法
}

```

创建类 MyRunnable.java 代码如下:

```

package myrunnable;

import entity.Userinfo;

public class MyRunnable implements Runnable {

    private Userinfo userinfo;

    public MyRunnable(Userinfo userinfo) {
        super();
        this.userinfo = userinfo;
    }

    @Override
    public void run() {
        userinfo.setUsername("usernameValue");
        userinfo.setPassword("passwordValue");
    }
}

```

创建类 Test.java 代码如下:

```

package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

```

```

import myrunnable.MyRunnable;
import entity.Userinfo;

public class Test {

    FutureTask abc;

    public static void main(String[] args) {
        try {
            Userinfo userinfo = new Userinfo();
            MyRunnable myrunnable = new MyRunnable(userinfo);

            ThreadPoolExecutor pool = new ThreadPoolExecutor(10, 10, 10,
                TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
            Future<Userinfo> future = pool.submit(myrunnable, userinfo);
            System.out.println("begin time=" + System.currentTimeMillis());
            userinfo = future.get();
            System.out.println("get value " + userinfo.getUsername() + " "
                + userinfo.getPassword());
            System.out.println("end time=" + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

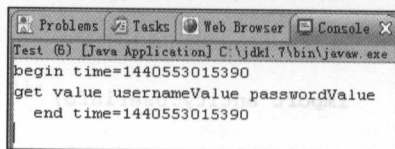


图 5-4 运行结果

程序运行结果如图 5-4 所示。

从控制台打印的结果来看，接口 Future 的实现类是 FutureTask.java，接口 Future 的继承与实现结构如图 5-5 所示。

其中就包括实现类 FutureTask.java，接口 Future 的声明结构如图 5-6 所示。

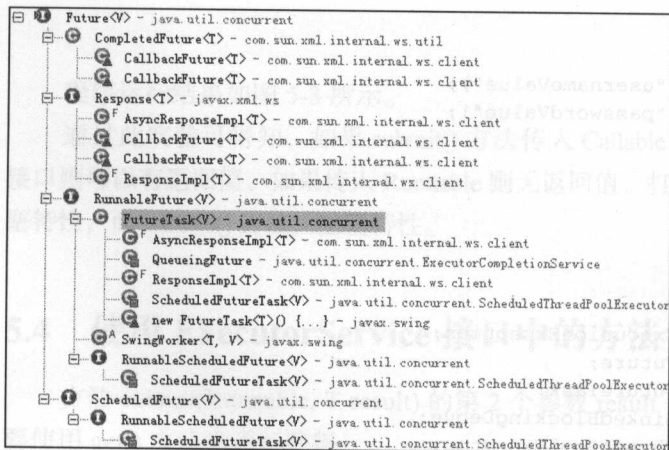


图 5-5 接口 Future 结构

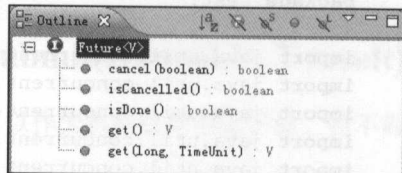


图 5-6 接口 Future 方法结构

接口 Future 中的方法不多, 其中 get() 及 isDone() 方法在前面已经介绍过, 下面继续学习其他的方法。

5.5 方法 cancel(boolean mayInterruptIfRunning) 和 isCancelled() 的使用

方法 cancel(boolean mayInterruptIfRunning) 的参数 mayInterruptIfRunning 的作用是: 如果线程正在运行则是否中断正在运行的线程, 在代码中需要使用 if (Thread.currentThread().isInterrupted()) 进行配合。

方法 cancel() 的返回值代表发送取消任务的命令是否成功完成。

创建实验用的项目 future_callable_4, 类 MyCallable.java 代码如下:

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        Thread.sleep(2000);
        return "我的年龄是100";
    }
}
```

创建 Test.java 类代码如下:

```
package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallable;

public class Test {

    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        MyCallable callable = new MyCallable();
        ExecutorService executor = new ThreadPoolExecutor(50,
            Integer.MAX_VALUE, 5, TimeUnit.SECONDS,
            new LinkedBlockingDeque<Runnable>());
    }
}
```



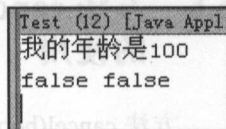
```

Future<String> future = executor.submit(callable);
System.out.println(future.get());
System.out.println(future.cancel(true) + " " + future.isCancelled());
}
}

```

程序运行结果如图 5-7 所示。

从打印的结果来看，线程任务已经运行完毕，线程对象已经销毁，所以方法 `cancel()` 的返回值是 `false`，代表发送取消的命令并没有成功。如果线程任务没有执行完毕，则调用 `cancel()` 方法会是什么效果呢？



```

Test (12) [Java Appl]
我的年龄是100
false false

```

图 5-7 运行结果

创建名称为 `test8` 的项目，创建类 `MyCallable.java` 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        Thread.sleep(2000);
        return "我的年龄是 100";
    }

}

```

类 `Test.java` 代码如下：

```

package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallable;

public class Test {

    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        MyCallable callable = new MyCallable();
        ExecutorService executor = new ThreadPoolExecutor(50,
            Integer.MAX_VALUE, 5, TimeUnit.SECONDS,
            new LinkedBlockingDeque<Runnable>());
        Future<String> future = executor.submit(callable);
        System.out.println(future.cancel(true) + " " + future.isCancelled());
    }

}

```

程序运行结果如图 5-8 所示。

任务在没有运行完成之前执行了 `cancel()` 方法返回为 `true`，代表成功发送取消的命令。

前面介绍过参数 `mayInterruptIfRunning` 具有中断线程的作用，并且需要结合代码 `if (Thread.currentThread().isInterrupted())` 来进行实现，此效果在新的项目中进行测试。

创建名称为 `test9` 的项目，类 `MyCallable.java` 代码如下：

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        int i = 0;
        while (i == 0) {
            if (Thread.currentThread().isInterrupted()) {
                throw new InterruptedException();
            }
            System.out.println("正在运行中");
        }
        return "我的年龄是 100";
    }
}
```

类 `Test.java` 代码如下：

```
package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallable;

public class Test {

    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        MyCallable callable = new MyCallable();
        ExecutorService executor = new ThreadPoolExecutor(50,
            Integer.MAX_VALUE, 5, TimeUnit.SECONDS,
            new LinkedBlockingDeque<Runnable>());
        Future<String> future = executor.submit(callable);
    }
}
```

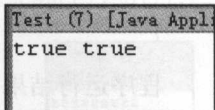


图 5-8 运行结果

```
Thread.sleep(4000);
System.out.println(future.cancel(true) + " " + future.isCancelled());
```

程序运行结果如图 5-9 所示。

线程被成功中断，不再打印“正在运行中”字符，cancel() 方法返回 true 代表发送中断线程的命令发送成功。

那如果不结合 if (Thread.currentThread().isInterrupted()) 代码会是什么效果呢？

创建名称为 test10 的项目，类 MyCallable.java 代码如下：

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        int i = 0;
        while (i == 0) {
            System.out.println("zzzzzzzzzz");
        }
        return "我的年龄是 100";
    }
}
```



图 5-9 运行结果

类 Test.java 代码如下：

```
package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallable;

public class Test {

    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        MyCallable callable = new MyCallable();
        ExecutorService executor = new ThreadPoolExecutor(50,
            Integer.MAX_VALUE, 5, TimeUnit.SECONDS,
            new LinkedBlockingDeque<Runnable>());
```

```

import java.util.concurrent.*;

Future<String> future = executor.submit(callable);
Thread.sleep(4000);
System.out.println(future.cancel(true) + " " + future.isCancelled());
}
}

```

程序运行结果如图 5-10 所示。

从打印的结果来看，线程并未中断运行，返回 true 代表发送中断线程的命令是成功的，但是否中断要取决于有没有 if (Thread.currentThread().isInterrupted()) 代码。

如果方法 cancel() 传入的参数是 false 有什么效果呢？

创建名称为 test12 的项目，类 MyCallable.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        try {
            while (1 == 1) {
                if (Thread.currentThread().isInterrupted() == true) {
                    throw new InterruptedException();
                }
                System.out.println(Thread.currentThread().getName() + " "
                    + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "我的年龄是 100";
    }
}

```

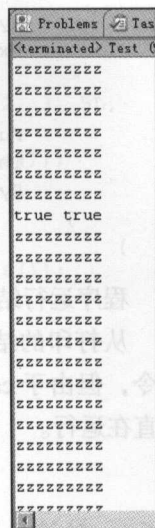


图 5-10 运行结果

类 Test.java 代码如下：

```

package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallable;

```

```

public class Test {
    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        MyCallable callable = new MyCallable();
        ExecutorService executor = new ThreadPoolExecutor(50,
            Integer.MAX_VALUE, 5, TimeUnit.SECONDS,
            new LinkedBlockingDeque<Runnable>());
        Future<String> future = executor.submit(callable);
        Thread.sleep(3000);
        System.out.println(future.cancel(false) + " " + future.isCancelled());
    }
}

```

程序运行结果如图 5-11 所示。

从打印的结果来看，输出了一个 true，则代表成功发送取消的命令，但由于 cancel() 方法的参数值是 false，所以线程并没有中断一直在运行。

```

pool-1-thread-1 1435113877406
pool-1-thread-1 1435113877406
pool-1-thread-1 1435113877406
pool-1-thread-1 1435113877406
pool-1-thread-1 1435113877406
pool-1-thread-1 1435113877406
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968
true true
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968
pool-1-thread-1 1435113877968

```

图 5-11 运行结果

5.6 方法 get(long timeout, TimeUnit unit) 的使用

方法 get(long timeout, TimeUnit unit) 的作用是在指定的最大时间内等待获得返回值。

创建实验用的项目 future_callable_5，类 MyCallable.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {
    public String call() throws Exception {
        Thread.sleep(10000);
        System.out.println("sleep 10 秒执行完了！");
        return "anyString";
    }
}

```

创建类 Run1.java 代码如下：

```

package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

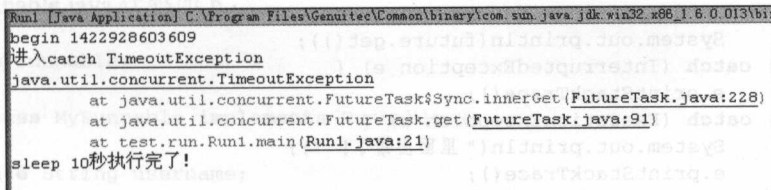
```

```
import mycallable.MyCallable;

public class Run1 {

    public static void main(String[] args) {
        try {
            MyCallable callable = new MyCallable();
            ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 3, 5L,
                TimeUnit.SECONDS, new LinkedBlockingDeque());
            System.out.println("begin " + System.currentTimeMillis());
            Future<String> future = executor.submit(callable);
            System.out.println("返回值 " + future.get(5,
                TimeUnit.SECONDS));
            System.out.println(" end " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            System.out.println("进入 catch InterruptedException");
            e.printStackTrace();
        } catch (ExecutionException e) {
            System.out.println("进入 catch ExecutionException");
            e.printStackTrace();
        } catch (TimeoutException e) {
            System.out.println("进入 catch TimeoutException");
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 5-12 所示。



```
Run1 [Java Application] C:\Program Files\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin
begin 1422928603609
进入 catch TimeoutException
java.util.concurrent.TimeoutException
    at java.util.concurrent.FutureTask$Sync.innerGet(FutureTask.java:228)
    at java.util.concurrent.FutureTask.get(FutureTask.java:91)
    at test.run.Run1.main(Run1.java:21)
sleep 10秒执行完了!
```

图 5-12 程序超时出现异常

5.7 异常的处理

接口 Callable 任务在执行时有可能出现异常，那在 Callable 中异常是如何处理的呢？

创建实验用的项目 future_callable_6，创建类 MyCallable.java 代码如下：

```
package extcallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {
    private String number;
```



```

public MyCallable(String number) {
    super();
    this.number = number;
}

@Override
public String call() throws Exception {
    Integer.parseInt("a");
    return "我是高洪岩" + number;
}
}

```

创建类 Run.java 代码如下：

```

package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import extcallable.MyCallable;

public class Run {

    public static void main(String[] args) {
        try {
            MyCallable callable = new MyCallable("1");
            ExecutorService executorRef =
                Executors.newCachedThreadPool();
            Future<String> future = executorRef.submit(callable);
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            System.out.println("里面出错了！");
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 5-13 所示。

```

里面出错了!
java.util.concurrent.ExecutionException: java.lang.NumberFormatException: For input string: "a"
    at java.util.concurrent.FutureTask$Sync.innerGet(FutureTask.java:222)
    at java.util.concurrent.FutureTask.get(FutureTask.java:83)
    at test.run.Run.main(Run.java:17)
Caused by: java.lang.NumberFormatException: For input string: "a"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at extcallable.MyCallable.call(MyCallable.java:15)
    at extcallable.MyCallable.call(MyCallable.java:1)
    at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:303)
    at java.util.concurrent.FutureTask.run(FutureTask.java:138)
    at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
    at java.lang.Thread.run(Thread.java:619)

```

图 5-13 异常被捕获

控制台显示的信息说明如果出现异常，则进入 catch 语句，不再继续执行 get() 方法了，这与通过 for 循环调用 get() 方法时的效果是一样的，不再继续执行 for 循环，直接进入 catch 语句块。

5.8 自定义拒绝策略 RejectedExecutionHandler 接口的使用

接口 RejectedExecutionHandler 的主要作用是当线程池关闭后依然有任务要执行时，可以实现一些处理。

创建实验用的项目 RejectedExecutionHandlerTest，类 MyRejectedExecutionHandler.java 代码如下：

```
package executionhandler;

import java.util.concurrent.FutureTask;
import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadPoolExecutor;

public class MyRejectedExecutionHandler implements RejectedExecutionHandler {
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        System.out.println(((FutureTask) r).toString() + " 被拒绝!");
    }
}
```

类 MyRunnable.java 代码如下：

```
package myrunnable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    public void run() {
        System.out.println(username + " 在运行!");
    }
}
```

创建类 Run.java 代码如下：

```
package test.run;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```

import java.util.concurrent.ThreadPoolExecutor;

import myrunnable.MyRunnable;
import executionhandler.MyRejectedExecutionHandler;

public class Run {

    public static void main(String[] args) {

        ExecutorService service = Executors.newCachedThreadPool();
        ThreadPoolExecutor executor = (ThreadPoolExecutor) service;
        executor.setRejectedExecutionHandler(new MyRejectedExecutionHandler());
        service.submit(new MyRunnable("A"));
        service.submit(new MyRunnable("B"));
        service.submit(new MyRunnable("C"));
        executor.shutdown();
        service.submit(new MyRunnable("D"));

    }
}

```

程序运行结果如图 5-14 所示。

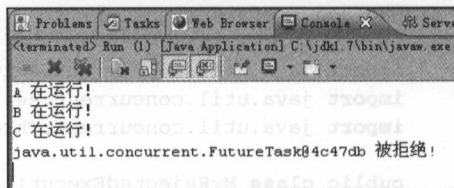


图 5-14 被拒绝运行

5.9 方法 execute() 与 submit() 的区别

方法 execute() 没有返回值，而 submit() 方法可以有返回值。

方法 execute() 在默认的情况下异常直接抛出，不能捕获，但可以通过自定义 Thread-Factory 的方式进行捕获，而 submit() 方法在默认的情况下，可以 catch Execution-Exception 捕获异常。

(1) 有 / 无返回值的测试

创建名称为 execute_submit_diff 的项目，类 Test1.java 代码如下：

```

package test;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Test1 {

    public static void main(String[] args) {
        try {

```

```

package test;

import java.util.concurrent.*;

public class Test1 {
    ExecutorService executor = new ThreadPoolExecutor(50,
        Integer.MAX_VALUE, 5, TimeUnit.SECONDS,
        new LinkedBlockingDeque<Runnable>());
    executor.execute(new Runnable() {
        @Override
        public void run() {
            System.out.println("execute() 方法执行了, 没有返回值");
        }
    });
    Future future = executor.submit(new Callable<String>() {
        @Override
        public String call() throws Exception {
            System.out.println("submit() 方法执行了, 有返回值");
            return "我是返回值";
        }
    });
    System.out.println(future.get());
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
}
}

```

Test1 (19) [Java Application] C:\jdk1.7\bin
 execute()方法执行了, 没有返回值
 submit()方法执行了, 有返回值
 我是返回值

图 5-15 运行结果

程序运行结果如图 5-15 所示。

从运行结果来看, `execute()` 没有返回值, 而 `submit()` 方法具有返回值的功能。

(2) `execute()` 出现异常后直接打印堆栈信息, 而 `submit()` 方法可以捕获

继续, 创建类 `Test2.java` 代码如下:

```

package test;

import java.util.concurrent.*;

public class Test2 {
    public static void main(String[] args) {
        ExecutorService executor = new ThreadPoolExecutor(50,
            Integer.MAX_VALUE, 5, TimeUnit.SECONDS,
            new LinkedBlockingDeque<Runnable>());
        executor.execute(new Runnable() {
            @Override
            public void run() {
                Integer.parseInt("a");
            }
        });
    }
}

```

程序运行后的效果如图 5-18 所示。

图 5-17 方法调用结果
 图 5-18 方法调用结果
 图 5-19 方法调用结果

程序运行后的效果如图 5-16 所示。

```
Exception in thread "pool-1-thread-1" java.lang.NumberFormatException: For input string: "a"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:492)
at java.lang.Integer.parseInt(Integer.java:527)
at test.Test2$1.run(Test2.java:17)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:745)
```

图 5-16 方法 execute() 直接抛出异常而不能捕获

继续，创建类 Test3.java 代码如下：

```
package test;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Test3 {

    public static void main(String[] args) {
        try {
            ExecutorService executor = new ThreadPoolExecutor(50,
                Integer.MAX_VALUE, 5, TimeUnit.SECONDS,
                new LinkedBlockingDeque<Runnable>());
            Future future = executor.submit(new Callable<String>() {
                @Override
                public String call() throws Exception {
                    Integer.parseInt("a");
                    return "我是返回值 ";
                }
            });
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
            System.out.println("能捕获异常");
        }
    }
}
```

程序运行后的效果如图 5-17 所示。

(3) execute() 方法异常也可以捕获

继续，创建类 Test4.java 代码如下：

```

package test;

import java.lang.Thread.UncaughtExceptionHandler;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Test4 {

    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(50,
            Integer.MAX_VALUE, 5, TimeUnit.SECONDS,
            new LinkedBlockingDeque<Runnable>());
        executor.setThreadFactory(new ThreadFactory() {
            @Override
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r);
                t.setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
                    @Override
                    public void uncaughtException(Thread t, Throwable e) {
                        System.out.println("execute() 方法通过使用自定义");
                        System.out.println("ThreadFactory 也能捕获异常了");
                        e.printStackTrace();
                    }
                });
                return t;
            }
        });
        executor.execute(new Runnable() {
            @Override
            public void run() {
                Integer.parseInt("a");
            }
        });
    }
}

```

```

java.util.concurrent.ExecutionException: java.lang.NumberFormatException: For input string: "a"
    at java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.util.concurrent.FutureTask.get(FutureTask.java:188)
    at test.Test3.main(Test3.java:25)
Caused by: java.lang.NumberFormatException: For input string: "a"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.parseInt(Integer.java:527)
    at test.Test3$1.call(Test3.java:21)
    at test.Test3$1.call(Test3.java:1)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)
能捕获异常

```

图 5-17 方法 submit() 出现异常并能捕获

程序运行后的效果如图 5-18 所示。


```

execute()方法通过使用自定义
ThreadFactory也能捕获异常了
java.lang.NumberFormatException: For input string: "a"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.parseInt(Integer.java:527)
    at test.Test432.run(Test4.java:33)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)

```

图 5-18 方法 execute() 也能捕获异常了

5.10 验证 Future 的缺点

类 FutureTask 声明如图 5-19 所示。

接口 Future 的实现类是 FutureTask.java，而且在使用线程池时，默认的情况下也是使用 FutureTask.java 类作为接口 Future 的实现类，也就是说，如果在使用 Future 与 Callable 的情况下，使用 Future 接口也就是在使用 FutureTask.java 类。

```

java.util.concurrent
类 FutureTask<V>

java.lang.Object
└ java.util.concurrent.FutureTask<V>

类型参数:
    V - 此 FutureTask 的 get 方法所返回的结果类型。

所有已实现的接口:
    Runnable, Future<V>, RunnableFuture<V>

```

图 5-19 声明结构

Callable 接口与 Runnable 接口在对比时主要的优点是，Callable 接口可以通过 Future 取得返回值。但需要注意的是，Future 接口调用 get() 方法取得处理的结果值时是阻塞性的，也就是如果调用 Future 对象的 get() 方法时，任务尚未执行完成，则调用 get() 方法时一直阻塞到此任务完成时为止。如果是这样的效果，则前面先执行的任务一旦耗时很多，则后面的任务调用 get() 方法就呈阻塞状态，也就是排队进行等待，大大影响运行效率。也就是主线程并不能保证首先获得的是最先完成任务的返回值，这就是 Future 的缺点，影响效率。

创建验证用的项目，名称为 futureLast，类 MyCallable.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    private String username;
    private long sleepValue;

    public MyCallable(String username, long sleepValue) {
        super();
        this.username = username;
        this.sleepValue = sleepValue;
    }

    @Override
    public String call() throws Exception {

```

```

        System.out.println(username);
        Thread.sleep(sleepValue);
        return "return " + username;
    }
}

```

类 Test.java 代码如下:

```

package test;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallable;

public class Test {

    public static void main(String[] args) {
        try {
            MyCallable callable1 = new MyCallable("username1", 5000);
            MyCallable callable2 = new MyCallable("username2", 4000);
            MyCallable callable3 = new MyCallable("username3", 3000);
            MyCallable callable4 = new MyCallable("username4", 2000);
            MyCallable callable5 = new MyCallable("username5", 1000);

            List<Callable> callableList = new ArrayList<Callable>();
            callableList.add(callable1);
            callableList.add(callable2);
            callableList.add(callable3);
            callableList.add(callable4);
            callableList.add(callable5);

            List<Future> futureList = new ArrayList<Future>();

            ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 5,
                TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
            for (int i = 0; i < 5; i++) {
                futureList.add(executor.submit(callableList.get(i)));
            }

            System.out
                .println("run first time= " + System.currentTimeMillis());
            for (int i = 0; i < 5; i++) {
                System.out.println(futureList.get(i).get() + " "
                    + System.currentTimeMillis());
            }
        }
    }
}

```

图 6-3 类 ExecutorCompletionService 的构造方法

```

        // 按顺序打印的效果
        // 说明一个 Future 对应指定的一个 Callable
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
}

```

程序运行结果如图 5-20 所示。

上面的示例就是 Future 接口的缺点，根据这个特性，JDK1.5 提供了 CompletionService 接口可以解决这个问题，关于此接口的使用请参看第 6 章。

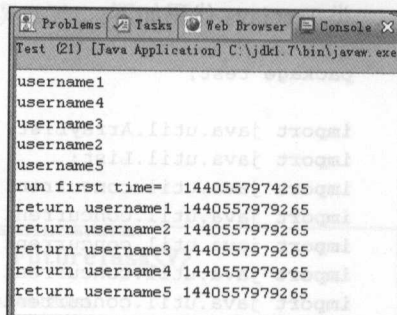


图 5-20 方法 get() 呈阻塞状态

5.11 本章总结

本章对 Future 和 Callable 接口中的全部 API 进行了介绍，这两个接口的优点就是从线程中返回数据以便进行后期的处理，但 FutureTask 类也有其自身的缺点，就是阻塞性，解决这个缺点请参看第 6 章。

CompletionService 的使用

接口 `CompletionService` 的功能是以异步的方式一边生产新的任务，一边处理已完成任务的结果，这样可以将执行任务与处理任务分离开来进行处理。使用 `submit` 执行任务，使用 `take` 取得已完成的任务，并按照完成这些任务的时间顺序处理它们的结果。

接口 `CompletionService` 的核心方法如图 6-1 所示。

```

equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
poll() : Future - CompletionService
poll(long timeout, TimeUnit unit) : Future - CompletionService
submit(Callable task) : Future - CompletionService
submit(Runnable task, Object result) : Future - CompletionService
take() : Future - CompletionService
toString() : String - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object

```

图 6-1 接口 `CompletionService` 的核心方法

6.1 CompletionService 介绍

接口 `CompletionService` 的结构如图 6-2 所示。

接口 `CompletionService` 的结构比较简洁，仅有一个实现类 `ExecutorCompletionService`，该类的构造方法如图 6-3 所示。

```

java.util.concurrent
接口 CompletionService<V>
所有已知实现类:
    ExecutorCompletionService

```

图 6-2 接口 `CompletionService` 的结构

构造方法摘要

<code>ExecutorCompletionService(Executor executor)</code>	使用为执行基本任务而提供的执行程序创建一个 <code>ExecutorCompletionService</code> ，并将 <code>LinkedBlockingQueue</code> 作为完成队列。
<code>ExecutorCompletionService(Executor executor, BlockingQueue<Future<V>> completionQueue)</code>	使用为执行基本任务而提供的执行程序创建一个 <code>ExecutorCompletionService</code> ，并将所提供的队列作为其完成队列。

图 6-3 类 `ExecutorCompletionService` 的构造方法

从构造方法的声明中可以发现，类 `ExecutorCompletionService` 需要依赖于 `Executor` 对象，大部分的实现也就是使用线程池 `ThreadPoolExecutor` 对象。

6.2 使用 `CompletionService` 解决 `Future` 的缺点

第 5 章演示过接口 `Future` 具有阻塞同步性，这样的代码运行效率会大打折扣，接口 `CompletionService` 可以解决这样的问题。

在本示例中使用 `CompletionService` 接口中的 `take()` 方法，它的主要作用就是取得 `Future` 对象，方法声明结构如下：

```
public Future<V> take() throws InterruptedException
```

创建测试用的项目 `ExecutorCompletionService_0`，创建类 `MyCallable.java` 代码如下：

```
package mycallable;
```

```
import java.util.concurrent.Callable;
```

```
public class MyCallable implements Callable<String> {
```

```
    private String username;
```

```
    private long sleepValue;
```

```
    public MyCallable(String username, long sleepValue) {
```

```
        super();
```

```
        this.username = username;
```

```
        this.sleepValue = sleepValue;
```

```
    }
```

```
    @Override
```

```
    public String call() throws Exception {
```

```
        System.out.println(username);
```

```
        Thread.sleep(sleepValue);
```

```
        return "return " + username;
```

```
    }
```

```
}
```

类 `Test1.java` 代码如下：

```
package test;
```

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
import java.util.concurrent.Callable;
```

```
import java.util.concurrent.CompletionService;
```

```
import java.util.concurrent.ExecutionException;
```

```
import java.util.concurrent.ExecutorCompletionService;
```

```
import java.util.concurrent.LinkedBlockingDeque;
```

```
import java.util.concurrent.ThreadPoolExecutor;
```

```

import java.util.concurrent.TimeUnit;

import mycallable.MyCallable;

public class Test1 {

    public static void main(String[] args) {
        try {
            MyCallable callable1 = new MyCallable("username1", 5000);
            MyCallable callable2 = new MyCallable("username2", 4000);
            MyCallable callable3 = new MyCallable("username3", 3000);
            MyCallable callable4 = new MyCallable("username4", 2000);
            MyCallable callable5 = new MyCallable("username5", 1000);

            List<Callable> callableList = new ArrayList<Callable>();
            callableList.add(callable1);
            callableList.add(callable2);
            callableList.add(callable3);
            callableList.add(callable4);
            callableList.add(callable5);

            ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 5,
                TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
            CompletionService csRef = new ExecutorCompletionService(executor);

            for (int i = 0; i < 5; i++) {
                csRef.submit(callableList.get(i));
            }

            for (int i = 0; i < 5; i++) {
                System.out.println("等待打印第 " + (i + 1) + " 个返回值");
                System.out.println(csRef.take().get());
            }
            // 按乱序打印的效果
            // 说明一个 Future 对应当前先执行完的 Callable 任务
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 6-4 所示。

从打印的结果来看，CompletionService 完全解决了 Future 阻塞的特性，也就是使用 CompletionService 接口后，哪个任务先执行完，哪个任务的返回值就先打印。

在 CompletionService 接口中如果当前没有任务被执行完，则 csRef.take().get() 方法还是呈阻塞特性。

类 Test2.java 代码如下：

```

Test1 [Java Application] C:\
username1
username5
username4
username3
username2
等待打印第1个返回值
return username5
等待打印第2个返回值
return username4
等待打印第3个返回值
return username3
等待打印第4个返回值
return username2
等待打印第5个返回值
return username1

```

图 6-4 运行结果


```

package test;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallable;

public class Test2 {

    public static void main(String[] args) {
        try {
            MyCallable callable1 = new MyCallable("username1", 5000);
            MyCallable callable2 = new MyCallable("username2", 4000);
            MyCallable callable3 = new MyCallable("username3", 3000);
            MyCallable callable4 = new MyCallable("username4", 2000);
            MyCallable callable5 = new MyCallable("username5", 1000);

            List<Callable> callableList = new ArrayList<Callable>();
            callableList.add(callable1);
            callableList.add(callable2);
            callableList.add(callable3);
            callableList.add(callable4);
            callableList.add(callable5);

            ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 5,
                TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>());
            CompletionService csRef = new ExecutorCompletionService(executor);

            for (int i = 0; i < 5; i++) {
                csRef.submit(callableList.get(i));
            }

            for (int i = 0; i < 6; i++) {
                System.out.println("等待打印第 " + (i + 1) + " 个返回值");
                System.out.println(csRef.take().get());
            }

            // 按乱序打印的效果
            // 说明一个 Future 对应当前先执行完的 Callable 任务
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

循环次数变成6次。

程序运行结果如图 6-5 所示。

6.3 使用 take() 方法

方法 take() 取得最先完成任务的 Future 对象，谁执行时间最短谁最先返回。

创建测试用的项目 ExecutorCompletionService_1，类 Run.java 代码如下：

```
package test.run;

import java.util.concurrent.Callable;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Run {

    public static void main(String[] args) {
        try {
            // take 方法: 获取并移除表示下一个已完成任务的 Future，如果目前不存在这样的任务，则等待。
            ExecutorService executorService = Executors.newCachedThreadPool();
            CompletionService csRef = new ExecutorCompletionService(
                executorService);
            for (int i = 0; i < 10; i++) {
                csRef.submit(new Callable<String>() {
                    public String call() throws Exception {
                        long sleepValue = (int) (Math.random() * 1000);
                        System.out.println("sleep=" + sleepValue + " "
                            + Thread.currentThread().getName());
                        Thread.sleep(sleepValue);
                        return "高洪岩: " + sleepValue + " "
                            + Thread.currentThread().getName();
                    }
                });
            }
            for (int i = 0; i < 10; i++) {
                System.out.println(csRef.take().get());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

```
Test2 [Java Application] C:\
username1
username4
username5
username3
username2
等待打印第1个返回值
return username5
等待打印第2个返回值
return username4
等待打印第3个返回值
return username3
等待打印第4个返回值
return username2
等待打印第5个返回值
return username1
等待打印第6个返回值
```

图 6-5 永远在阻塞
永远在等待

程序运行结果如图 6-6 所示。

从运行结果来看，方法 `take()` 是按任务执行的速度，从快到慢的顺序获得 `Future` 对象，因为打印的时间是从小到大。

6.4 使用 `poll()` 方法

方法 `poll()` 的作用是获取并移除表示下一个已完成任务的 `Future`，如果不存在这样的任务，则返回 `null`，方法 `poll()` 无阻塞的效果。

创建测试用的项目 `ExecutorCompletionService_2`，类 `Run.java` 代码如下：

```
package test.run;

import java.util.concurrent.Callable;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Run {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        CompletionService csRef = new ExecutorCompletionService(executorService);
        for (int i = 0; i < 1; i++) {
            csRef.submit(new Callable<String>() {
                public String call() throws Exception {
                    Thread.sleep(3000);
                    System.out.println("3 秒过后了");
                    return "高洪岩 3s";
                }
            });
        }
        for (int i = 0; i < 1; i++) {
            System.out.println(csRef.poll());
        }
    }
}
```

程序运行结果如图 6-7 所示。

从运行结果来看，方法 `poll()` 返回的 `Future` 为 `null`，因为当前没有任何已完成任



图 6-6 运行结果

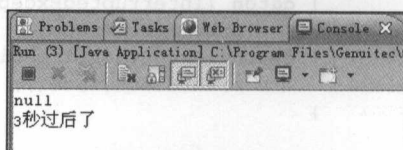


图 6-7 运行结果

务的 Future 对象，所以返回为 null，通过此实验证明 poll() 方法不像 take() 方法具有阻塞的效果。

6.5 使用 poll(long timeout, TimeUnit unit) 方法

方法 Future<V> poll(long timeout, TimeUnit unit) 的作用是等待指定的 timeout 时间，在 timeout 时间之内获取到值时立即向下继续执行，如果超时也立即向下执行。

创建测试用的项目 ExecutorCompletionService_3，类 MyCallableA.java 代码如下：

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {
    public String call() throws Exception {
        Thread.sleep(5000);
        System.out.println("MyCallableA " + System.currentTimeMillis());
        return "returnA";
    }
}
```

类 MyCallableB.java 代码如下：

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {
    public String call() throws Exception {
        Thread.sleep(10000);
        System.out.println("MyCallableB " + System.currentTimeMillis());
        return "returnB";
    }
}
```

类 Run1.java 代码如下：

```
package test.run;

import java.util.concurrent.CompletionService;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run1 {
```

```

public static void main(String[] args) {
    MyCallableA callableA = new MyCallableA();
    MyCallableB callableB = new MyCallableB();

    Executor executor = Executors.newCachedThreadPool();
    CompletionService csRef = new ExecutorCompletionService(executor);
    csRef.submit(callableA);
    csRef.submit(callableB);

    for (int i = 0; i < 2; i++) {
        System.out.println("zzzzzzzzzzzz" + " " + csRef.poll());
    }
    System.out.println("main end!");
}

```

程序运行结果如图 6-8 所示。

返回 2 个 null 值，因为任务未执行完毕。

类 Run2.java 代码如下：

```

zzzzzzzzzzzz null
zzzzzzzzzzzz null
main end!
MyCallableA 1423099150921
MyCallableB 1423099155921

```

图 6-8 运行结果

```

package test.run;

import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run2 {

    public static void main(String[] args) {

        try {
            MyCallableA callableA = new MyCallableA();
            MyCallableB callableB = new MyCallableB();

            Executor executor = Executors.newCachedThreadPool();
            CompletionService csRef = new ExecutorCompletionService(executor);
            csRef.submit(callableA);
            csRef.submit(callableB);

            for (int i = 0; i < 2; i++) {
                System.out.println("zzzzzzzzzzzz" + " "
                    + csRef.poll(6, TimeUnit.SECONDS).get());
                System.out.println("X");
            }
            System.out.println("main end!");
        }
    }
}

```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
}

```

程序运行结果如图 6-9 所示。

返回 2 个值，因为一共等待了 12 秒。

类 Run3.java 代码如下：

```
package test.run;
```

```

import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run3 {

    public static void main(String[] args) {

        try {
            MyCallableA callableA = new MyCallableA();
            MyCallableB callableB = new MyCallableB();

            Executor executor = Executors.newCachedThreadPool();
            CompletionService csRef = new ExecutorCompletionService(executor);
            csRef.submit(callableA);
            csRef.submit(callableB);

            System.out.println("zzzzzzzzzzzz" + " "
                + csRef.poll(4, TimeUnit.SECONDS) + " "
                + System.currentTimeMillis());
            System.out.println("X");
            System.out.println("zzzzzzzzzzzz" + " "
                + csRef.poll(2, TimeUnit.SECONDS).get() + " "
                + System.currentTimeMillis());
            System.out.println("X");
            System.out.println("zzzzzzzzzzzz" + " "
                + csRef.poll(5, TimeUnit.SECONDS).get() + " "
                + System.currentTimeMillis());
            System.out.println("X");

            System.out.println("main end!");
        }
    }
}

```

```

MyCallableA 1423099235640
zzzzzzzzzzzz returnA
X
MyCallableB 1423099240640
zzzzzzzzzzzz returnB
X
main end!

```

图 6-9 运行结果


```

    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
}

```

程序运行结果如图 6-10 所示。

在第 4 秒时打印的返回值是 null，因为还未到达 5 秒，他 2 次打印都正确得到了返回值，因为任务已经完成。

类 Run4.java 代码如下：

```
package test.run;
```

```

import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run4 {

    public static void main(String[] args) {

        try {
            MyCallableA callableA = new MyCallableA();
            MyCallableB callableB = new MyCallableB();

            Executor executor = Executors.newCachedThreadPool();
            CompletionService csRef = new ExecutorCompletionService(executor);
            csRef.submit(callableA);
            csRef.submit(callableB);

            System.out.println("zzzzzzzzzzzz" + " "
                + csRef.poll(4, TimeUnit.SECONDS) + " "
                + System.currentTimeMillis());
            System.out.println("X");
            System.out.println("zzzzzzzzzzzz" + " "
                + csRef.poll(7, TimeUnit.SECONDS).get() + " "
                + System.currentTimeMillis());
            System.out.println("X");

            System.out.println("main end!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {

```

```

zzzzzzzzzzzz null 1423099292875
X
MyCallableA 1423099293875
zzzzzzzzzzzz returnA 1423099293875
X
MyCallableB 1423099298875
zzzzzzzzzzzz returnB 1423099298875
X
main end!

```

图 6-10 运行结果

```

        e.printStackTrace();
    }
}

```

程序运行结果如图 6-11 所示。

从控制台打印的结果来看，方法 poll() 中 timeout 的值如果小于任务执行的时间，则返回值就是 null。

```

zzzzzzzzzz null 1423099405343
X
MyCallableA 1423099406328
zzzzzzzzzz returnA 1423099406328
X
main end!
MyCallableB 1423099411328

```

6.6 类 CompletionService 与异常

图 6-11 运行结果

使用 CompletionService 执行任务的过程中不可避免会出现各种情况的异常，下面来实验一下 CompletionService 对异常处理的流程。

创建实验用的项目 ExecutorCompletionService_errorHandle，类 MyCallableA.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableA begin " + System.currentTimeMillis());
        Thread.sleep(1000);
        System.out.println("MyCallableA end " + System.currentTimeMillis());
        return "returnA";
    }
}

```

类 MyCallableB.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableB begin " + System.currentTimeMillis());
        Thread.sleep(5000);
        int i = 0;
        if (i == 0) {
            throw new Exception(" 抛出异常! ");
        }
        System.out.println("MyCallableB end " + System.currentTimeMillis());
        return "returnB";
    }
}

```

运行类 Run1.java 代码如下：

```
package test.run;

import java.util.concurrent.CompletionService;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run1 {

    public static void main(String[] args) {
        try {
            MyCallableA callableA = new MyCallableA();
            MyCallableB callableB = new MyCallableB();

            Executor executor = Executors.newSingleThreadExecutor();
            CompletionService csRef = new ExecutorCompletionService(executor);
            csRef.submit(callableA);
            csRef.submit(callableB);

            for (int i = 0; i < 2; i++) {
                System.out.println("zzzzzzzzzzzz" + " " + csRef.take());
            }
            System.out.println("main end!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
MyCallableA begin 1440728988750
MyCallableA end 1440728989750
MyCallableB begin 1440728989750
zzzzzzzzzzzz java.util.concurrent.FutureTask@659db7
zzzzzzzzzzzz java.util.concurrent.FutureTask@16be68f
main end!
```

图 6-12 打印 FutureTask 对象

程序运行结果如图 6-12 所示。

上面的示例虽然 MyCallableB.java 出现异常，但并没有调用 FutureTask 类的 get() 方法，所以不出现异常。

类 Run2.java 代码如下：

```
package test.run;

import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run2 {
```

```

public static void main(String[] args) {
    try {
        MyCallableA callableA = new MyCallableA();
        MyCallableB callableB = new MyCallableB();

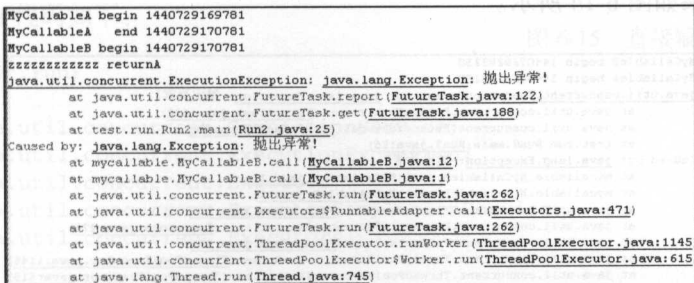
        Executor executor = Executors.newSingleThreadExecutor();
        CompletionService csRef = new ExecutorCompletionService(executor);
        csRef.submit(callableA); // 先执行的 A
        csRef.submit(callableB); // 后执行的 B

        for (int i = 0; i < 2; i++) {
            System.out.println("zzzzzzzzzzzz" + " " + csRef.take().get());
        }

        System.out.println("main end!");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
}

```

程序运行结果如图 6-13 所示。



```

MyCallableA begin 1440729169781
MyCallableA end 1440729170781
MyCallableB begin 1440729170781
zzzzzzzzzzzz returnA
java.util.concurrent.ExecutionException: java.lang.Exception: 抛出异常!
    at java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.util.concurrent.FutureTask.get(FutureTask.java:188)
    at test.run.Run2.main(Run2.java:25)
Caused by: java.lang.Exception: 抛出异常!
    at mycallable.MyCallableB.call(MyCallableB.java:12)
    at mycallable.MyCallableB.call(MyCallableB.java:1)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)

```

图 6-13 任务 A 正常任务 B 出现异常

任务 A 执行时间较少,也并未出现异常,正确打印任务 A 的返回值,任务 B 出现异常。类 Run3.java 代码如下:

```

package test.run;

import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

```

```

public class Run3 {

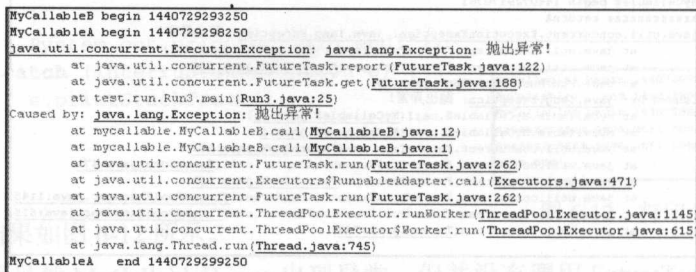
    public static void main(String[] args) {
        try {
            MyCallableA callableA = new MyCallableA();
            MyCallableB callableB = new MyCallableB();

            Executor executor = Executors.newSingleThreadExecutor();
            CompletionService csRef = new ExecutorCompletionService(executor);
            csRef.submit(callableB); // 先执行 B
            csRef.submit(callableA); // 后执行 A

            for (int i = 0; i < 2; i++) {
                System.out.println("zzzzzzzzzzzz" + " " + csRef.take().get());
            }
            System.out.println("main end!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 6-14 所示。



```

MyCallableB begin 1440729293250
MyCallableA begin 1440729298250
java.util.concurrent.ExecutionException: java.lang.Exception: 抛出异常!
    at java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.util.concurrent.FutureTask.get(FutureTask.java:188)
    at test.run.Run3.main(Run3.java:25)
Caused by: java.lang.Exception: 抛出异常!
    at mycallable.MyCallableB.call(MyCallableB.java:12)
    at mycallable.MyCallableB.call(MyCallableB.java:1)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)
MyCallableA end 1440729299250

```

图 6-14 任务 B 出现异常任务 A 并未输出

类 Run4.java 代码如下：

```

package test.run;

import java.util.concurrent.CompletionService;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run4 {

```

```

public static void main(String[] args) {
    try {
        MyCallableA callableA = new MyCallableA();
        MyCallableB callableB = new MyCallableB();

        Executor executor = Executors.newSingleThreadExecutor();
        CompletionService csRef = new ExecutorCompletionService(executor);
        csRef.submit(callableA);
        csRef.submit(callableB);

        for (int i = 0; i < 2; i++) {
            System.out.println("zzzzzzzzzzzz" + " " + csRef.poll());
        }

        Thread.sleep(6000);
        System.out.println("A处" + " " + csRef.poll());
        System.out.println("B处" + " " + csRef.poll());
        System.out.println("main end!");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

程序运行结果如图 6-15 所示。

类 Run5.java 代码如下：

```

===== null
===== null
MyCallableA begin 1440729381156
MyCallableA end 1440729382156
MyCallableB begin 1440729382156
A处 java.util.concurrent.FutureTask$11c2b67
B处 java.util.concurrent.FutureTask$659db7
main end!

```

图 6-15 直接输出 FutureTask 对象

```
package test.run;
```

```

import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run5 {

    public static void main(String[] args) {
        try {
            MyCallableA callableA = new MyCallableA();
            MyCallableB callableB = new MyCallableB();

            Executor executor = Executors.newSingleThreadExecutor();
            CompletionService csRef = new ExecutorCompletionService(executor);
            csRef.submit(callableA);    // 先执行 A
            csRef.submit(callableB);    // 后执行 B

            for (int i = 0; i < 2; i++) {
                System.out.println("zzzzzzzzzzzz" + " " + csRef.poll());
            }

            Thread.sleep(6000);
        }
    }
}

```



```

        System.out.println("A 处 " + " " + csRef.poll().get());
        System.out.println("B 处 " + " " + csRef.poll().get());
        System.out.println("main end!");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
}

```

程序运行结果如图 6-16 所示。

```

zzzzzzzzzz null
zzzzzzzzzz null
MyCallableA begin 1440729472515
MyCallableA end 1440729473515
MyCallableB begin 1440729473515
A处 returnA
java.util.concurrent.ExecutionException: java.lang.Exception: 抛出异常!
    at java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.util.concurrent.FutureTask.get(FutureTask.java:188)
    at test.run.Run5.main(Run5.java:29)
Caused by: java.lang.Exception: 抛出异常!
    at mycallable.MyCallableB.call(MyCallableB.java:12)
    at mycallable.MyCallableB.call(MyCallableB.java:1)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)

```

图 6-16 打印任务 A 返回值任务 B 出现异常

类 Run6.java 代码如下：

```

package test.run;

import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run6 {

    public static void main(String[] args) {
        try {
            MyCallableA callableA = new MyCallableA();
            MyCallableB callableB = new MyCallableB();

            Executor executor = Executors.newSingleThreadExecutor();
            CompletionService csRef = new ExecutorCompletionService(executor);
            csRef.submit(callableB);    // 先执行 B
            csRef.submit(callableA);    // 后执行 A
        }
    }
}

```

```

for (int i = 0; i < 2; i++) {
    System.out.println("zzzzzzzzzzzz" + " " + csRef.poll());
}
Thread.sleep(6000);
System.out.println("A处" + " " + csRef.poll().get());
System.out.println("B处" + " " + csRef.poll().get());
System.out.println("main end!");
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
}
}

```

程序运行结果如图 6-17 所示。

```

zzzzzzzzzz null
zzzzzzzzzz null
MyCallableB begin 1440729519703
MyCallableA begin 1440729524703
java.util.concurrent.ExecutionException: java.lang.Exception: 抛出异常!
MyCallableA end 1440729525703
    at java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.util.concurrent.FutureTask.get(FutureTask.java:198)
    at test.run.Run6.main(Run6.java:28)
Caused by: java.lang.Exception: 抛出异常!
    at mycallable.MyCallableB.call(MyCallableB.java:12)
    at mycallable.MyCallableB.call(MyCallableB.java:1)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)

```

图 6-17 任务 A 并未打印任务 B 抛出异常

6.7 方法 Future<V> submit(Runnable task, V result) 的测试

参数 V 是 submit() 方法的返回值。

创建测试用的项目 ExecutorCompletionService_4, 实体类 Userinfo.java 代码如下:

```
package entity;
```

```
public class Userinfo {
```

```
    private String username;
    private String password;
```

```
    public Userinfo() {
        super();
    }
```

```
    public Userinfo(String username, String password) {
        super();
    }
```

```

        this.username = username;
        this.password = password;
    }

    // get 和 set 方法
}

```

类 MyRunnable.java 代码如下:

```

package myrunnable;

import entity.UserInfo;

public class MyRunnable implements Runnable {
    private UserInfo userinfo;

    public MyRunnable(UserInfo userinfo) {
        super();
        this.userinfo = userinfo;
    }

    @Override
    public void run() {
        userinfo.setUsername("usernameValue");
        userinfo.setPassword("passwordValue");
        System.out.println("run 运行了 ");
    }
}

```

类 Run1.java 代码如下:

```

package test.run;

import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import myrunnable.MyRunnable;
import entity.UserInfo;

public class Run1 {

    public static void main(String[] args) {
        try {
            UserInfo userinfo = new UserInfo();
            MyRunnable myRunnable = new MyRunnable(userinfo);

            Executor executor = Executors.newCachedThreadPool();
            CompletionService csRef = new ExecutorCompletionService(executor);

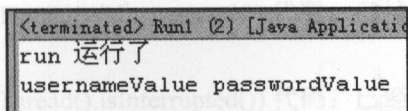
```

```

Future<Userinfo> future = csRef.submit(myRunnable, userinfo);
System.out.println(future.get().getUsername() + " "
    + future.get().getPassword());
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
}
}

```

程序运行结果如图 6-18 所示。



```

<terminated> Run1 (2) [Java Applicatio...
run 运行了
usernameValue passwordValue

```

图 6-18 成功返回 userinfo 实体类

6.8 本章总结

接口 CompletionService 完全可以避免 FutureTask 类阻塞的缺点，可更加有效地处理 Future 的返回值，也就是哪个任务先执行完，CompletionService 就先取得这个任务的返回值再处理。

创建类 MyCallable1.java 代码如下：

图 7-1 接口 ExecutorService 的核心方法

7.1 在 ThreadPoolExecutor 中使用 ExecutorService 的方法

方法 invokeAny() 和 invokeAll() 具有阻塞特性。

Chapter 7 第 7 章

接口 ExecutorService 的方法使用

接口 `ExecutorService` 中有很多工具方法，在前面章节中已经介绍过一部分，那么在本章中将对剩余方法的功能逐一进行介绍，以帮助大家更好地掌握 `ExecutorService` 接口。`ExecutorService` 接口中的方法有些与 `Future` 和 `Callable` 有关，所以 `Future` 和 `Callable` 的使用是学习 `ExecutorService` 接口中方法的基础。

接口 `ExecutorService` 的核心方法如图 7-1 所示。

```
● awaitTermination(long timeout, TimeUnit unit) : boolean - ExecutorService
● equals(Object obj) : boolean - Object
● execute(Runnable command) : void - Executor
● getClass() : Class<?> - Object
● hashCode() : int - Object
● invokeAll(Collection<? extends Callable<T>> tasks) : List<Future<T>> - ExecutorService
● invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) : List<Future<T>> - ExecutorService
● invokeAny(Collection<? extends Callable<T>> tasks) : T - ExecutorService
● invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) : T - ExecutorService
● isShutdown() : boolean - ExecutorService
● isTerminated() : boolean - ExecutorService
● notify() : void - Object
● notifyAll() : void - Object
● shutdown() : void - ExecutorService
● shutdownNow() : List<Runnable> - ExecutorService
● submit(Callable<T> task) : Future<T> - ExecutorService
● submit(Runnable task) : Future<?> - ExecutorService
● submit(Runnable task, T result) : Future<T> - ExecutorService
● toString() : String - Object
● wait() : void - Object
● wait(long timeout) : void - Object
● wait(long timeout, int nanos) : void - Object
```

图 7-1 接口 `ExecutorService` 的核心方法

7.1 在 `ThreadPoolExecutor` 中使用 `ExecutorService` 中的方法

方法 `invokeAny()` 和 `invokeAll()` 具有阻塞特性。

方法 `invokeAny()` 取得第一个完成任务的结果值, 当第一个任务执行完成后, 会调用 `interrupt()` 方法将其他任务中断, 所以在这些任务中可以结合 `if (Thread.currentThread().isInterrupted()==true)` 代码来决定任务是否继续运行。

方法 `invokeAll()` 等全部线程任务执行完毕后, 取得全部完成任务的结果值。

7.2 方法 `invokeAny(Collection tasks)` 的使用与 `interrupt`

此实验验证方法 `invokeAny()` 的确是取得第一个完成任务的结果值, 但在这个过程中出现两种情况:

1) 无 `if (Thread.currentThread().isInterrupted())` 代码: 已经获得第一个运行的结果值后, 其他线程继续运行。

2) 有 `if (Thread.currentThread().isInterrupted())` 代码: 已经获得第一个运行的结果值后, 其他线程如果使用 `throw new InterruptedException()` 代码则这些线程中断, 虽然 `throw` 抛出了异常, 但在 `main` 线程中并不能捕获异常。如果想捕获异常, 则需要在 `Callable` 中使用 `try-catch` 显式进行捕获。

创建测试用的项目 `ExecutorService_invokeAny_1`, 创建类 `MyCallableA.java` 代码如下:

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableA begin " + System.currentTimeMillis());
        for (int i = 0; i < 123456; i++) {
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableA " + (i + 1));
        }
        System.out.println("MyCallableA end " + System.currentTimeMillis());
        return "returnA";
    }
}
```

创建类 `MyCallableB1.java` 代码如下:

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB1 implements Callable<String> {
```



```

public String call() throws Exception {
    System.out.println("MyCallableB begin " + System.currentTimeMillis());
    for (int i = 0; i < 223456; i++) {
        Math.random();
        Math.random();
        Math.random();
        System.out.println("MyCallableB " + (i + 1));
    }
    System.out.println("MyCallableB end " + System.currentTimeMillis());
    return "returnB";
}
}

```

创建类 MyCallableB2.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB2 implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableB begin " + System.currentTimeMillis());
        for (int i = 0; i < 223456; i++) {
            if (Thread.currentThread().isInterrupted() == false) {
                Math.random();
                Math.random();
                Math.random();
                System.out.println("MyCallableB " + (i + 1));
            } else {
                System.out.println("***** 抛出异常中断了");
                throw new InterruptedException();
            }
        }
        System.out.println("MyCallableB end " + System.currentTimeMillis());
        return "returnB";
    }
}

```

创建类 Run1.java 代码如下：

```

package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB1;

```

```

public class Run1 {
    public static void main(String[] args) {
        try {
            List list = new ArrayList();
            list.add(new MyCallableA());
            list.add(new MyCallableB1());

            ExecutorService executor = Executors.newCachedThreadPool();
            // invokeAny 只取得最先完成任务的结果值
            String getValueA = executor.invokeAny(list);
            System.out.println("===== " + getValueA);
            System.out.println("ZZZZZZZZZZZZZZZZZZ");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 7-2 所示。

虽然方法 `invokeAny()` 已经取得 `returnA` 的值，但线程 B 还在继续运行中，直到运行完毕。

类 `Run2.java` 代码如下：

```

package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB2;

public class Run2 {
    public static void main(String[] args) {
        try {
            List list = new ArrayList();
            list.add(new MyCallableA());
            list.add(new MyCallableB2());

            ExecutorService executor = Executors.newCachedThreadPool();
            // invokeAny 只取得最先完成任务的结果值
            String getValueA = executor.invokeAny(list);
            System.out.println("===== " + getValueA);
            System.out.println("ZZZZZZZZZZZZZZZZZZ");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

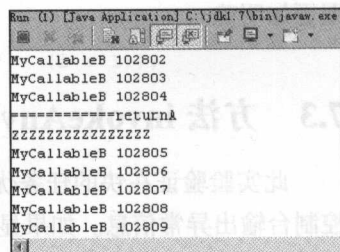


图 7-2 运行结果

```

    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
}

```

使用 `MyCallableB2.java` 类，运行结果如图 7-3 所示。

控制台打印的结果说明线程 A 执行完毕后，线程池将线程 B 设置为中断 `interrupte` 状态，而线程 B 可以自定义对中断 `interrupte` 状态进行处理，也就是可以决定是否使用 `Thread.currentThread().isInterrupted()` 结合 `throw new InterruptedException()` 的代码。

如果使用 `Thread.currentThread().isInterrupted()` 结合 `throw new InterruptedException()` 的代码说明对线程 B 进行中断的意图更加明确。

```

Run2 (2) [Java Application] C:\jdk1.7\bin\java.exe
MyCallableB 30933
MyCallableB 30934
MyCallableB 30935
MyCallableA end 1423106079171
MyCallableB 30936
MyCallableB 30937
MyCallableB 30938
MyCallableB 30939
MyCallableB 30940
MyCallableB 30941
MyCallableB 30942
MyCallableB 30943
MyCallableB 30944
MyCallableB 30945
*****抛出异常中断了
*****returnA
ZZZZZZZZZZZZZZZZZZZZ

```

图 7-3 运行结果

7.3 方法 `invokeAny()` 与执行慢的任务异常

此实验验证在快的任务优先执行完毕后，执行慢的任务出现异常时，默认情况下不会在控制台输出异常信息。如果显式使用 `try-catch` 语句块则可以自定义捕获异常。

创建测试用的项目 `test3`，创建类 `MyCallableA.java` 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    @Override
    public String call() throws Exception {
        System.out.println("MyCallableA " + Thread.currentThread().getName()
            + " begin " + System.currentTimeMillis());
        for (int i = 0; i < 123456; i++) {
            String newString = new String();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableA 在运行中=" + (i + 1));
        }
        System.out.println("MyCallableA " + Thread.currentThread().getName()
            + " end " + System.currentTimeMillis());
        return "returnA";
    }
}

```

创建类 MyCallableB.java 代码如下:

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    @Override
    public String call() throws Exception {
        System.out.println("MyCallableB " + Thread.currentThread().getName()
            + " begin " + System.currentTimeMillis());
        for (int i = 0; i < 193456; i++) {
            String newString = new String();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableB 在运行中 =" + (i + 1));
        }
        if (1 == 1) {
            System.out.println("xxxxxxx= 中断了");
            throw new NullPointerException();
        }
        System.out.println("MyCallableB_END "
            + Thread.currentThread().getName() + " end "
            + System.currentTimeMillis());
        return "returnB";
    }
}
```

创建类 Run.java 代码如下:

```
package test;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run {

    public static void main(String[] args) {

        try {
            List list = new ArrayList();
            list.add(new MyCallableA());
```

```

        list.add(new MyCallableB());

        ExecutorService service = Executors.newCachedThreadPool();
        String getString = service.invokeAny(list);
        System.out.println("zzzz=" + getString);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
}
}

```

程序运行结果如图 7-4 所示。

```

MyCallableB 在运行中=64520
MyCallableB 在运行中=64521
MyCallableB 在运行中=64522
MyCallableB 在运行中=64523
zzzz=returnA
MyCallableB 在运行中=64524
MyCallableB 在运行中=64525
MyCallableB 在运行中=64526

```

```

MyCallableB 在运行中=193455
MyCallableB 在运行中=193456
xxxxxxxx=中断了

```

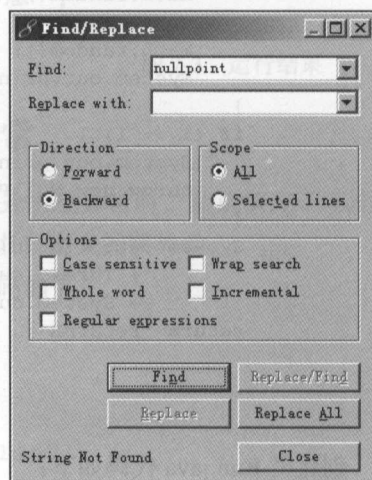


图 7-4 运行结果

程序运行的结果：成功取得 returnA 字符串，线程 B 中断了，但抛出的空指针异常却没有在控制台输出。

如果想要在 Callable 中捕获异常信息，则需要显式地添加 try-catch 语句块。下面实验一下这个测试。更改 MyCallableB.java 类代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    @Override
    public String call() throws Exception {
        try {

```

```

        System.out.println("MyCallableB "
            + Thread.currentThread().getName() + " begin "
            + System.currentTimeMillis());
        for (int i = 0; i < 193456; i++) {
            String newString = new String();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableB 在运行中=" + (i + 1));
        }
        if (1 == 1) {
            System.out.println("xxxxxxx= 中断了");
            throw new NullPointerException();
        }
        System.out.println("MyCallableB_END "
            + Thread.currentThread().getName() + " end "
            + System.currentTimeMillis());
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println(e.getMessage() + " 通过显式 try-catch 捕获异常了");
        throw e;
    }
    return "returnB";
}
}

```

继续更改 Run.java 类代码如下:

```

package test;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run {

    public static void main(String[] args) {

        try {
            List list = new ArrayList();
            list.add(new MyCallableA());
            list.add(new MyCallableB());

            ExecutorService service = Executors.newCachedThreadPool();
            String getString = service.invokeAny(list);

```



```

        System.out.println("zzzz="+getString());
    } catch (InterruptedException e) {
        e.printStackTrace();
        System.out.println("mainA");
    } catch (ExecutionException e) {
        e.printStackTrace();
        System.out.println("mainB");
    }
}
}
}

```

程序运行结果如图 7-5 所示。

```

MyCallableB 在运行中=89573
MyCallableB 在运行中=89574
zzzz=returnA
MyCallableB 在运行中=89575
MyvCallableB 在运行中=89576

MyCallableB 在运行中=193453
MyCallableB 在运行中=193454
MyCallableB 在运行中=193455
MyCallableB 在运行中=193456
xxxxxxx=中断了
java.lang.NullPointerException
    at mycallable.MyCallableB.call(MyCallableB.java:24)
    at mycallable.MyCallableB.call(MyCallableB.java:1)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)
null 通过显式try-catch捕获异常了

```

图 7-5 运行结果

从运行结果来看，加入显式的 try-catch 语句块可以捕获异常信息，但抛出去的异常在 main() 方法中却没有得到捕获，也就是字符串 mainA 和 mainB 没有被打印，也就说明子线程出现异常时是不影响 main 线程的主流程的，此实验在项目名称为 test4 中有源代码。

7.4 方法 invokeAny() 与执行快的任务异常

此实验验证在执行快的任务出现异常时，在默认情况下是不在控制台输出异常信息的，除非显式使用 try-catch 捕获，而等待执行慢的任务返回的结果值。

先出现异常而不影响后面任务的取值的原理是在源代码中一直判断有没有正确的返回的值，如果直到最后都没有获得返回值则抛出异常，这个异常是最后出现的异常，比如 A、B、C 这 3 个任务一起被执行，都出现了异常，则最终的异常就是在最后出现的异常。此结论可以查看 AbstractExecutorService 类的方法 private <T> T doInvokeAny(Collection<? extends Callable<T>> tasks, boolean timed, long nanos) 中的源代码。

创建测试用的项目 test6，创建类 MyCallableA.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    @Override
    public String call() throws Exception {
        System.out.println("MyCallableA " + Thread.currentThread().getName()
            + " begin " + System.currentTimeMillis());
        for (int i = 0; i < 123456; i++) {
            String newString = new String();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableA 在运行中 = " + (i + 1));
        }
        if (1 == 1) {
            System.out.println("xxxxxxx= 中断了");
            throw new NullPointerException();
        }
        System.out.println("MyCallableAEND " + Thread.currentThread().getName()
            + " end " + System.currentTimeMillis());
        return "returnA";
    }
}

```

创建类 MyCallableB.java 代码如下:

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    @Override
    public String call() throws Exception {
        System.out.println("MyCallableB " + Thread.currentThread().getName()
            + " begin " + System.currentTimeMillis());
        for (int i = 0; i < 193456; i++) {
            String newString = new String();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableB 在运行中 = " + (i + 1));
        }
        System.out.println("MyCallableB " + Thread.currentThread().getName()
            + " end " + System.currentTimeMillis());
    }
}

```

```

        return "returnB";
    }
}

```

创建类 Run.java 代码如下：

```

package test;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import mycallable.MyCallableB;
import mycallable.MyCallableA;

public class Run {

    public static void main(String[] args) {

        try {
            List list = new ArrayList();
            list.add(new MyCallableB());
            list.add(new MyCallableA());

            ExecutorService service = Executors.newCachedThreadPool();
            String getString = service.invokeAny(list);
            System.out.println("main 取得的返回值 = " + getString);
        } catch (InterruptedException e) {
            System.out.println("main InterruptedException");
            e.printStackTrace();
        } catch (ExecutionException e) {
            System.out.println("main Execution_Exception");
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 7-6 所示。

此实验说明线程 A 中断了，但并没有在控制台输出空指针异常相关的信息，所以最后对线程 B 的返回值进行输出，因为线程 B 并没有出现异常。

如果想捕获空指针异常使用显式 try-catch 的方式进行捕获即可，但在 MyCallableA 的 catch 块中如果不重新 throw 抛出异常，则主线程 main 不能取得 MyCallableB.java 任务的返回值，反而取得的是 MyCallableA 的返回值，也就是 MyCallableA 的异常状态并未上报，导致线程池认为 MyCallableA 是正确的，而未发现 MyCallableA 是一个异常的任务，也就是并未使关注点切换到下一个任务，即 MyCallableB 中。如果 MyCallableA 将异常再次抛出，则线程池认为 MyCallableA 出现了异常，则将关注点切换到 MyCallableB 中，取得的返回值是

MyCallableB 的。此实验的源代码在项目名称为 test7 中，关键类 MyCallableA.java 代码如下：

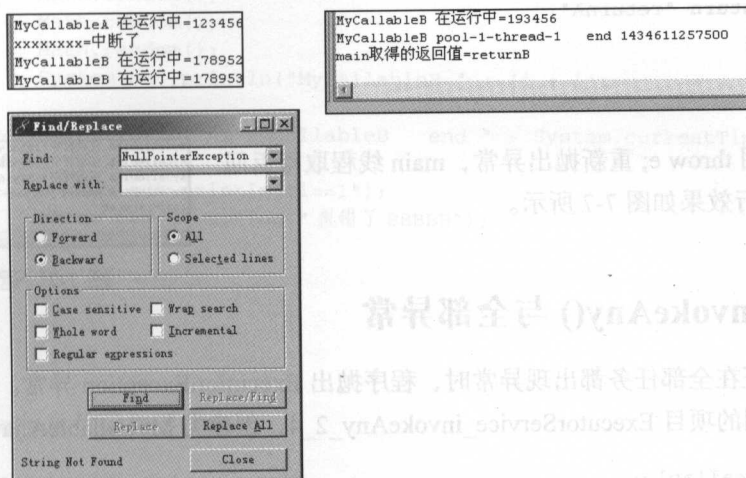


图 7-6 运行结果

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    @Override
    public String call() throws Exception {
        try {
            System.out.println("MyCallableA "
                + Thread.currentThread().getName() + " begin "
                + System.currentTimeMillis());
            for (int i = 0; i < 12345; i++) {
                String newString = new String();
                Math.random();
                Math.random();
                Math.random();
                Math.random();
                Math.random();
                System.out.println("MyCallableA 在运行中=" + (i + 1));
            }
            if (1 == 1) {
                System.out.println("xxxxxxx= 中断了");
                throw new NullPointerException();
            }
            System.out.println("MyCallableAEND "
                + Thread.currentThread().getName() + " end "
                + System.currentTimeMillis());
        } catch (Exception e) {
            System.out.println(e.getMessage() + " : 左边信息就是捕获的异常信息");
        }
    }
}
```

```

        throw e;
    }
    return "returnA";
}
}

```

代码中使用 `throw e`; 重新抛出异常, `main` 线程取得返回值 `returnB`, 运行效果如图 7-7 所示。

```

MyCallableB 在运行中=193454
MyCallableB 在运行中=193455
MyCallableB 在运行中=193456
MyCallableB pool-1-thread-1 end 1440899049265
zzzz=returnB

```

图 7-7 运行效果

7.5 方法 `invokeAny()` 与全部异常

此实验验证在全部任务都出现异常时, 程序抛出 `ExecutionException` 异常。

创建测试用的项目 `ExecutorService_invokeAny_2_4`, 创建类 `MyCallableA.java` 代码如下:

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableA begin " + System.currentTimeMillis());
        for (int i = 0; i < 123; i++) {
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableA " + (i + 1));
        }
        System.out.println("MyCallableA end " + System.currentTimeMillis());
        if (1 == 1) {
            System.out.println("1==1");
            throw new Exception(" 报错了 AAAAA");
        }
        return "returnA";
    }
}

```

创建类 `MyCallableB.java` 代码如下:

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableB begin " + System.currentTimeMillis());
    }
}

```

```

for (int i = 0; i < 123456; i++) {
    Math.random();
    Math.random();
    Math.random();
    System.out.println("MyCallableB " + (i + 1));
}
System.out.println("MyCallableB end " + System.currentTimeMillis());
if (1 == 1) {
    System.out.println("1==1");
    throw new Exception(" 报错了BBBBB");
}
return "returnB";
}
}

```

创建类 Run.java 代码如下:

```

package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run {

    public static void main(String[] args) {
        try {
            List list = new ArrayList();
            list.add(new MyCallableA());
            list.add(new MyCallableB());

            ExecutorService executor = Executors.newCachedThreadPool();
            System.out.println(executor);
            String getValueA = executor.invokeAny(list);
            System.out.println(" 返回值 " + getValueA);
            System.out.println("mainEND");
        } catch (InterruptedException e) {
            System.out.println(" 进入 catch InterruptedException");
            e.printStackTrace();
        } catch (ExecutionException e) {
            System.out.println(" 进入 catch ExecutionException");
            e.printStackTrace();
        }
    }
}

```


程序运行结果如图 7-8 所示。

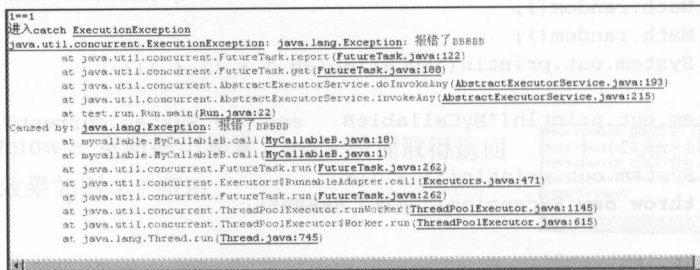


图 7-8 运行结果

都出现异常时返回最后一个异常并输出。

7.6 方法 invokeAny(CollectionTasks, timeout, TimeUnit) 超时的测试

方法 `<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)` 的主要作用就是在指定时间内取得第一个先执行完任务的结果值。

创建测试用的项目 `ExecutorService_invokeAny_6`，创建类 `MyCallableA.java` 代码如下：

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableA begin " + System.currentTimeMillis());
        for (int i = 0; i < 193456; i++) {
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableA i=" + (i + 1));
        }
        System.out.println("MyCallableA end " + System.currentTimeMillis());
        return "MyCallableAValue";
    }
}
```

创建类 `Run.java` 代码如下：

```

package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import mycallable.MyCallableA;

public class Run {

    public static void main(String[] args) {
        try {
            MyCallableA a = new MyCallableA();
            List callableList = new ArrayList();
            callableList.add(a);
            ExecutorService service = Executors.newCachedThreadPool();
            String getValue = service.invokeAny(callableList, 1,
                TimeUnit.SECONDS);
            System.out.println("=====" + getValue);
            System.out.println("zzzzzzzzzzzzzzzzzz");
        } catch (InterruptedException e) {
            System.out.println("进入 catch InterruptedException");
            e.printStackTrace();
        } catch (ExecutionException e) {
            System.out.println("进入 catch ExecutionException");
            e.printStackTrace();
        } catch (TimeoutException e) {
            System.out.println("进入 catch TimeoutException 超时了");
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 7-9 所示。

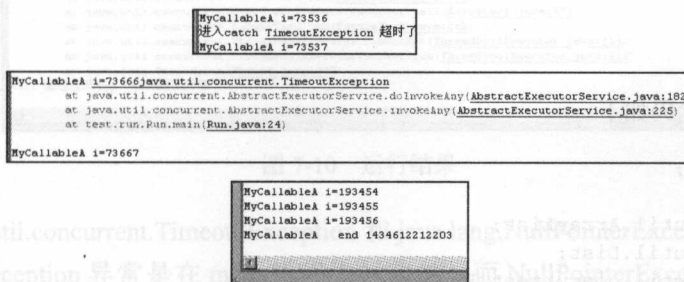


图 7-9 运行结果

在出现超时异常时，可以将 `if (Thread.currentThread().isInterrupted() == true)` 判断和 `throw new InterruptedException()` 结合以使线程中断执行。

如果只有一个任务被运行，这个任务既超时又出现了异常，运行结果会是什么样子呢？创建名称为 `ExecutorService_invokeAny_6_1` 的项目，类 `MyCallableA.java` 代码如下：

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    @Override
    public String call() throws Exception {
        try {
            System.out.println("MyCallableA "
                + Thread.currentThread().getName() + " begin "
                + System.currentTimeMillis());
            for (int i = 0; i < 193456; i++) {
                String newString = new String();
                Math.random();
                Math.random();
                Math.random();
                Math.random();
                Math.random();
                System.out.println("MyCallableA 在运行中=" + (i + 1));
                if (Thread.currentThread().isInterrupted() == true) {
                    System.out.println("xxxxxxx= 中断了");
                    throw new NullPointerException();
                }
            }
            System.out.println("MyCallableA "
                + Thread.currentThread().getName() + " end "
                + System.currentTimeMillis());
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println(e.getMessage() + " 通过显式 try-catch 捕获异常了");
            throw e;
        }
        return "returnB";
    }
}
```

类 `Run.java` 代码如下：

```
package test;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
```

```

import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import mycallable.MyCallableA;

public class Run {

    public static void main(String[] args) {

        try {
            List list = new ArrayList();
            list.add(new MyCallableA());

            ExecutorService service = Executors.newCachedThreadPool();
            String getString = service.invokeAny(list, 1, TimeUnit.SECONDS);
            System.out.println("zzzz=" + getString);
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.out.println("mainA");
        } catch (ExecutionException e) {
            e.printStackTrace();
            System.out.println("mainB");
        } catch (TimeoutException e) {
            e.printStackTrace();
            System.out.println("mainC");
        }
    }
}

```

程序运行结果如图 7-10 所示。

```

MyCallableA 在运行中=71114
MyCallableA 在运行中=71115
MyCallableA 在运行中=71116
MyCallableA 在运行中=71117
xxxxxxx=中断了
mainC
java.util.concurrent.TimeoutException
    at java.util.concurrent.AbstractExecutorService.doInvokeAny(AbstractExecutorService.java:182)
    at java.util.concurrent.AbstractExecutorService.invokeAny(AbstractExecutorService.java:225)
    at test.Run.main(Run.java:22)
java.lang.NullPointerException
    at mycallable.MyCallableA.call(MyCallableA.java:23)
    at mycallable.MyCallableA.call(MyCallableA.java:1)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)
null 通过try-catch捕获异常了

```

图 7-10 运行结果

异常 `java.util.concurrent.TimeoutException` 和 `java.lang.NullPointerException` 同时出现了，其中 `TimeoutException` 异常是在 `main` 线程中捕获的，而 `NullPointerException` 是在 `Callable` 中捕获的。

7.7 方法 invokeAll(Collection tasks) 全正确

方法 `invokeAll()` 会返回所有任务的执行结果，并且此方法执行的效果也是阻塞执行的，要把所有的结果都取回时再继续向下运行。

创建测试用的项目 `ExecutorService_invokeAll_1`，创建类 `CallableA.java` 代码如下：

```
package extthread;

import java.util.concurrent.Callable;

public class CallableA implements Callable<String> {

    @Override
    public String call() throws Exception {
        System.out.println(Thread.currentThread().getName() + " begin "
            + System.currentTimeMillis());
        Thread.sleep(5000);
        System.out.println(Thread.currentThread().getName() + " end "
            + System.currentTimeMillis());
        return "returnA";
    }
}
```

创建类 `CallableB.java` 代码如下：

```
package extthread;

import java.util.concurrent.Callable;

public class CallableB implements Callable<String> {

    @Override
    public String call() throws Exception {
        System.out.println(Thread.currentThread().getName() + " begin "
            + System.currentTimeMillis());
        Thread.sleep(8000);
        System.out.println(Thread.currentThread().getName() + " end "
            + System.currentTimeMillis());
        return "returnB";
    }
}
```

创建类 `Run.java` 代码如下：

```
package test;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```

import java.util.concurrent.Future;
import extthread.CallableA;
import extthread.CallableB;

public class Run {

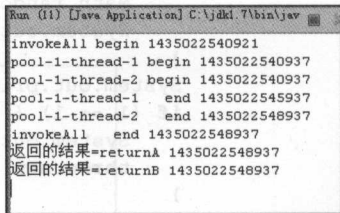
    public static void main(String[] args) {

        try {
            CallableA callableA = new CallableA();
            CallableB callableB = new CallableB();

            List<Callable<String>> list = new ArrayList<Callable<String>>();
            list.add(callableA);
            list.add(callableB);

            ExecutorService service = Executors.newCachedThreadPool();
            System.out.println("invokeAll begin " + System.currentTimeMillis());
            List<Future<String>> listFuture = service.invokeAll(list);
            System.out.println("invokeAll end " + System.currentTimeMillis());
            for (int i = 0; i < listFuture.size(); i++) {
                System.out.println(" 返回的结果=" + listFuture.get(i).get() + " "
                    + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.out.println("mainA");
        } catch (ExecutionException e) {
            e.printStackTrace();
            System.out.println("mainB");
        }
    }
}

```



```

Run (1) [Java Application] C:\jdk1.7\bin\jav
invokeAll begin 1435022540921
pool-1-thread-1 begin 1435022540937
pool-1-thread-2 begin 1435022540937
pool-1-thread-1 end 1435022545937
pool-1-thread-2 end 1435022548937
invokeAll end 1435022548937
返回的结果=returnA 1435022548937
返回的结果=returnB 1435022548937

```

图 7-11 运行结果

程序运行结果如图 7-11 所示。

方法 `invokeAll()` 执行的时间为 8 秒，由此可见其阻塞特性。

7.8 方法 `invokeAll(Collection tasks)` 快的正确慢的异常

在多个任务的过程中，执行任务快慢与运行时发生的异常也有一些联系。

创建测试用的项目 `ExecutorService_invokeAll_2`，创建类 `MyCallableA.java` 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

```



```

    public String call() throws Exception {
        System.out.println("MyCallableA begin " + System.currentTimeMillis());
        for (int i = 0; i < 123456; i++) {
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableA " + (i + 1));
        }
        System.out.println("MyCallableA end " + System.currentTimeMillis());
        return "returnA";
    }
}

```

创建类 MyCallableB.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableB begin " + System.currentTimeMillis());
        for (int i = 0; i < 223456; i++) {
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableB " + (i + 1));
        }
        System.out.println("MyCallableB end " + System.currentTimeMillis());
        if (1 == 1) {
            System.out.println("B 报错了");
            throw new Exception(" 出现异常 ");
        }
        return "returnB";
    }
}

```

创建类 Run.java 代码如下：

```

package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

```

```

public class Run {

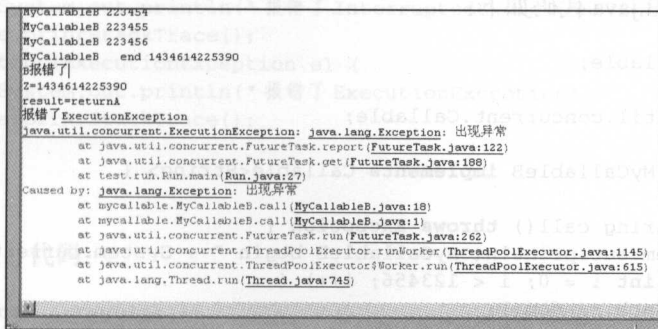
    public static void main(String[] args) {

        try {
            List list = new ArrayList();
            list.add(new MyCallableA());
            list.add(new MyCallableB());

            ExecutorService executor = Executors.newCachedThreadPool();
            System.out.println("Y=" + System.currentTimeMillis());
            List<Future<String>> listFuture = executor.invokeAll(list);
            System.out.println("Z=" + System.currentTimeMillis());
            for (int i = 0; i < listFuture.size(); i++) {
                System.out.println("result=" + listFuture.get(i).get());
            }
            System.out.println("mainEND");
        } catch (InterruptedException e) {
            System.out.println(" 报错了 InterruptedException");
            e.printStackTrace();
        } catch (ExecutionException e) {
            System.out.println(" 报错了 ExecutionException");
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 7-12 所示。



```

MyCallableB 223454
MyCallableB 223455
MyCallableB 223456
MyCallableB end 1434614225390
Y=1434614225390
Z=1434614225390
result=returnk
报错了 InterruptedException
报错了 ExecutionException
java.util.concurrent.ExecutionException: java.lang.Exception: 出现异常
    at java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.util.concurrent.FutureTask.get(FutureTask.java:180)
    at test.run.Run.main(Run.java:27)
Caused by: java.lang.Exception: 出现异常
    at mycallable.MyCallableB.call(MyCallableB.java:18)
    at mycallable.MyCallableB.call(MyCallableB.java:1)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)

```

图 7-12 运行结果

在控制台打印了“出现异常”字符串，说明 invokeAll() 方法对 Callable 抛出去的异常是可以处理的，由于在 main() 方法中直接进入了 catch 语句块，所以导致字符串 mainEND 也未打印出来。

如果使用 invokeAny() 方法而某一个任务正确地返回值时，则其他 Callable 抛出去的异常在 main() 方法中是不被处理的。

如果使用 invokeAny() 方法时都没有正确的返回值时，则说明最后 Callable 抛出去的异常在 main() 方法中是被处理了的。

7.9 方法 invokeAll(Collection tasks) 快的异常慢的正确

创建名称为 ExecutorService_invokeAll_2_2 的项目，类 MyCallableA.java 代码如下：

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableA begin " + System.currentTimeMillis());
        for (int i = 0; i < 123; i++) {
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableA " + (i + 1));
        }
        System.out.println("MyCallableA end " + System.currentTimeMillis());
        if (1 == 1) {
            System.out.println("A 报错了");
            throw new Exception(" 出现异常");
        }
        return "returnA";
    }
}
```

类 MyCallableB.java 代码如下：

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableB begin " + System.currentTimeMillis());
        for (int i = 0; i < 123456; i++) {
            Math.random();
            Math.random();
            Math.random();
            System.out.println("MyCallableB " + (i + 1));
        }
        System.out.println("MyCallableB end " + System.currentTimeMillis());
        return "returnB";
    }
}
```

类 Run.java 代码如下：

```
package test.run;
```

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run {

    public static void main(String[] args) {

        try {
            List list = new ArrayList();
            list.add(new MyCallableA());
            list.add(new MyCallableB());

            ExecutorService executor = Executors.newCachedThreadPool();
            System.out.println("Y=" + System.currentTimeMillis());
            List<Future<String>> listFuture = executor.invokeAll(list);
            System.out.println("Z=" + System.currentTimeMillis());
            for (int i = 0; i < listFuture.size(); i++) {
                System.out.println("result=" + listFuture.get(i).get());
            }
            System.out.println("mainEND");
        } catch (InterruptedException e) {
            System.out.println(" 报错了 InterruptedException");
            e.printStackTrace();
        } catch (ExecutionException e) {
            System.out.println(" 报错了 ExecutionException");
            e.printStackTrace();
        }
    }
}

```

需要注意的是，代码：

```

List list = new ArrayList();
list.add(new MyCallableA());
list.add(new MyCallableB());

```

向 list 添加 Callable 的顺序与运行结果有联系，在上面代码中先添加的 MyCallableA 任务，后添加的 MyCallableB 任务，则 main 方法中的代码为：

```

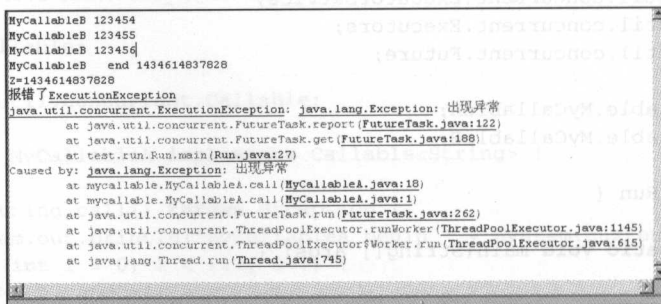
for (int i = 0; i < listFuture.size(); i++) {
    System.out.println("result=" + listFuture.get(i).get());
}

```

在第 1 次循环时取得的是 MyCallableA 的 Future 对象，由于 MyCallableA 速度快并且出现了异常，则在第一次 for 时出现 ExecutionException 异常，不再继续执行第 2 次循环，进入

main 方法中的 catch 语句块。

程序运行结果如图 7-13 所示。



```

MyCallableB 123454
MyCallableB 123455
MyCallableB 123456
MyCallableB end 1434614837828
Z=1434614837828
报错了ExecutionException
java.util.concurrent.ExecutionException: java.lang.Exception: 出现异常
    at java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.util.concurrent.FutureTask.get(FutureTask.java:188)
    at test.run.Run.main(Run.java:27)
Caused by: java.lang.Exception: 出现异常
    at mycallable.MyCallableA.call(MyCallableA.java:18)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)

```

图 7-13 运行结果

因为线程 A 出现了异常，在第一次 for 循环取得返回值时产生异常并退出 for 循环，也就导致了没有执行第 2 次 for 循环，所以在 main() 方法中没有获得线程 B 的返回值，程序流程直接进入 catch 语句块，导致 mainEND 字符串并没有输出。

使用 invokeAll() 方法如果全部任务都出现异常时，打印的结果与此示例效果一样。

7.10 方法 invokeAll(Collection tasks,long timeout,TimeUnit unit) 先慢后快

方法 invokeAll(Collection tasks,long timeout,TimeUnit unit) 的作用是如果全部任务在指定的时间内没有完成，则出现异常。

创建测试用的项目 test10_update_1，创建类 MyCallableA.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableA begin " + System.currentTimeMillis());
        for (int i = 0; i < 223456; i++) {
            Math.random();
            Math.random();
            System.out.println("MyCallableA i=" + (i + 1));
        }
        System.out.println("MyCallableA end " + System.currentTimeMillis());
        return "MyCallableAValue";
    }
}

```

创建类 MyCallableB.java 代码如下:

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableB begin " + System.currentTimeMillis());
        for (int i = 0; i < 10; i++) {
            Math.random();
            Math.random();
            System.out.println("MyCallableB i=" + (i + 1));
        }
        System.out.println("MyCallableB end " + System.currentTimeMillis());
        return "MyCallableBValue";
    }
}
```

创建类 Run.java 代码如下:

```
package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run {

    public static void main(String[] args) {

        try {
            MyCallableA a = new MyCallableA();
            MyCallableB b = new MyCallableB();

            List callableList = new ArrayList();
            callableList.add(a);
            callableList.add(b);

            ExecutorService service = Executors.newCachedThreadPool();
            System.out.println("X " + System.currentTimeMillis());
            List<Future<String>> listFuture = service.invokeAll(callableList,
                2, TimeUnit.SECONDS);
            System.out.println("Y " + System.currentTimeMillis());
        }
    }
}
```



```

        for (int i = 0; i < listFuture.size(); i++) {
            System.out.println("for 第 " + (i + 1) + " 次循环 ");
            System.out.println(listFuture.get(i).get());
        }
        System.out.println("Z " + System.currentTimeMillis());
    } catch (InterruptedException e) {
        System.out.println("进入 catch InterruptedException");
        e.printStackTrace();
    } catch (ExecutionException e) {
        System.out.println("进入 catch ExecutionException");
        e.printStackTrace();
    }
}
}

```

程序运行结果如图 7-14 所示。

```

MyCallableA i=104351
for 第1次循环
Exception in thread "main" MyCallableA i=104352
MyCallableA i=104353

java.util.concurrent.CancellationException
    at java.util.concurrent.FutureTask.report(FutureTask.java:121)
    at java.util.concurrent.FutureTask.get(FutureTask.java:188)
    at test.run.Run.main(Run.java:33)
MyCallableA i=104450

```

图 7-14 运行结果

由图 7-14 所示可知，并未打印 My-CallableAValue 和 MyCallableBValue 字符串。

从运行结果来看，使用 invokeAll() 方法出现超时后，调用 Future 对象的 get() 方法时出现的是 CancellationException 异常，而不是 invokeAny() 方法抛出来的 TimeoutException 异常。

7.11 方法 invokeAll(Collection tasks,long timeout,TimeUnit unit) 先快后慢

创建测试用的项目 test10_update_2，创建类 MyCallableA.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableA begin " + System.currentTimeMillis());
        for (int i = 0; i < 10; i++) {
            Math.random();
            Math.random();
            System.out.println("MyCallableA i=" + (i + 1));
        }
    }
}

```

```

        System.out.println("MyCallableA end " + System.currentTimeMillis());
        return "MyCallableAValue";
    }
}

```

创建类 MyCallableB.java 代码如下:

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableB begin " + System.currentTimeMillis());
        for (int i = 0; i < 223456; i++) {
            Math.random();
            Math.random();
            System.out.println("MyCallableB i=" + (i + 1));
        }
        System.out.println("MyCallableB end " + System.currentTimeMillis());
        return "MyCallableBValue";
    }
}

```

创建类 Run.java 代码如下:

```

package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run {

    public static void main(String[] args) {
        try {
            MyCallableA a = new MyCallableA();
            MyCallableB b = new MyCallableB();

            List callableList = new ArrayList();
            callableList.add(a);
            callableList.add(b);

            ExecutorService service = Executors.newCachedThreadPool();

```

```

        System.out.println("X " + System.currentTimeMillis());
        List<Future<String>> listFuture = service.invokeAll(callableList,
            2, TimeUnit.SECONDS);
        System.out.println("Y " + System.currentTimeMillis());
        for (int i = 0; i < listFuture.size(); i++) {
            System.out.println("for 第 " + (i + 1) + " 次循环 ");
            System.out.println(listFuture.get(i).get());
        }
        System.out.println("Z " + System.currentTimeMillis());
    } catch (InterruptedException e) {
        System.out.println("进入 catch InterruptedException");
        e.printStackTrace();
    } catch (ExecutionException e) {
        System.out.println("进入 catch ExecutionException");
        e.printStackTrace();
    }
}
}

```

程序运行结果如图 7-15 所示。

```

MyCallableB i=98154
for 第1次循环
MyCallableB i=98155
MyCallableB i=98156

MyCallableB i=98176
for 第2次循环
MyCallableB i=98177

MyCallableB i=98175
MyCallableAValue
MyCallableB i=98176

MyCallableB i=98288
MyCallableB i=98289java.util.concurrent.CancellationException
    at java.util.concurrent.FutureTask.report(FutureTask.java:121)
    at java.util.concurrent.FutureTask.get(FutureTask.java:188)
    at test.run.Run.main(Run.java:32)

```

图 7-15 运行结果

由图 7-15 所示可知，打印 MyCallable-AValue，但并未打印 MyCallableBValue 字符串，并出现一个异常。说明第一个 Future 没有超时，正常得到返回值，第二个 Future 由于超时没有正确得到返回值，调用 get() 方法时出现了 java.util.concurrent.CancellationException 异常。

7.12 方法 invokeAll(Collection tasks,long timeout,TimeUnit unit) 全慢

创建测试用的项目 test10_update_3，创建类 MyCallableA.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {

```

```

public String call() throws Exception {
    System.out.println("MyCallableA begin " + System.currentTimeMillis());
    for (int i = 0; i < 223456; i++) {
        Math.random();
        Math.random();
        System.out.println("MyCallableA i=" + (i + 1));
    }
    System.out.println("MyCallableA end " + System.currentTimeMillis());
    return "MyCallableAValue";
}
}

```

创建类 MyCallableB.java 代码如下:

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {

    public String call() throws Exception {
        System.out.println("MyCallableB begin " + System.currentTimeMillis());
        for (int i = 0; i < 223456; i++) {
            Math.random();
            Math.random();
            System.out.println("MyCallableB i=" + (i + 1));
        }
        System.out.println("MyCallableB end " + System.currentTimeMillis());
        return "MyCallableBValue";
    }
}

```

创建类 Run.java 代码如下:

```

package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run {

    public static void main(String[] args) {

```

```
try {
```

图 8-1 接口 ScheduledExecutorService 的核心方法列表

```

MyCallableA a = new MyCallableA();
MyCallableB b = new MyCallableB();

List callableList = new ArrayList();
callableList.add(a);
callableList.add(b);

ExecutorService service = Executors.newCachedThreadPool();
System.out.println("X " + System.currentTimeMillis());
List<Future<String>> listFuture = service.invokeAll(callableList,
    2, TimeUnit.SECONDS);
System.out.println("Y " + System.currentTimeMillis());
for (int i = 0; i < listFuture.size(); i++) {
    System.out.println(listFuture.get(i).get());
}
System.out.println("Z " + System.currentTimeMillis());
} catch (InterruptedException e) {
    System.out.println("进入 catch InterruptedException");
    e.printStackTrace();
} catch (ExecutionException e) {
    System.out.println("进入 catch ExecutionException");
    e.printStackTrace();
}
}
}

```

程序运行结果如图 7-16 所示。

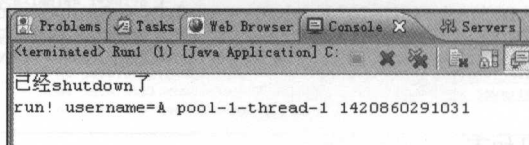


图 7-16 运行结果

由图 7-16 所示可知，并未打印 `MyCallableAValue` 和 `MyCallableBValue` 字符串，并出现一个异常，因为第 1 次 `for` 循环时就出现了异常，不再继续下一次 `for` 循环了。

7.13 本章总结

接口 `ExecutorService` 中的方法都以便携的方式去创建线程池，使用两个主要的方法 `invokeAny()` 和 `invokeAll()` 来取得第一个首先执行完任务的结果值，以及全部任务的结果值。

计划任务 ScheduledExecutorService 的使用

Java 中的计划任务 Timer 工具类提供了以计时器或计划任务的功能来实现按指定时间或时间间隔执行任务，但由于 Timer 工具类并不是以池 pool，而是以队列的方式来管理线程的，所以在高并发的情况下运行效率较低，在新版 JDK 中提供了 ScheduledExecutorService 对象来解决效率与定时任务的功能。

接口 ScheduledExecutorService 的核心方法列表如图 8-1 所示。

```

● awaitTermination(long timeout, TimeUnit unit) : boolean - ExecutorService
● equals(Object obj) : boolean - Object
● execute(Runnable command) : void - Executor
● getClass() : Class<?> - Object
● hashCode() : int - Object
● invokeAll(Collection<? extends Callable<T>> tasks) : List<Future<T>> - ExecutorService
● invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) : List<Future<T>> - ExecutorService
● invokeAny(Collection<? extends Callable<T>> tasks) : T - ExecutorService
● invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) : T - ExecutorService
● isShutdown() : boolean - ExecutorService
● isTerminated() : boolean - ExecutorService
● notify() : void - Object
● notifyAll() : void - Object
● schedule(Callable<V> callable, long delay, TimeUnit unit) : ScheduledFuture<V> - ScheduledExecutorService
● schedule(Runnable command, long delay, TimeUnit unit) : ScheduledFuture<?> - ScheduledExecutorService
● scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit) : ScheduledFuture<?> - ScheduledExecutorService
● scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit) : ScheduledFuture<?> - ScheduledExecutorService
● shutdown() : void - ExecutorService
● shutdownNow() : List<Runnable> - ExecutorService
● submit(Callable<T> task) : Future<T> - ExecutorService
● submit(Runnable task) : Future<?> - ExecutorService
● submit(Runnable task, T result) : Future<T> - ExecutorService
● toString() : String - Object
● wait() : void - Object
● wait(long timeout) : void - Object
● wait(long timeout, int nanos) : void - Object

```

图 8-1 接口 ScheduledExecutorService 的核心方法列表

8.1 ScheduledExecutorService 的使用

类 ScheduledExecutorService 的主要作用就是可以将定时任务与线程池功能结合使用。

此接口有 1 个主要的实现类，类实现关系如图 8-2 所示。
接口继承及实现关系如图 8-3 所示。

从图 8-3 中可以发现实现类 ScheduledThreadPool-
Executor 的父接口还是 Executor。

类 ScheduledThreadPoolExecutor 的父类是 ThreadPoolExecutor，继承关系如图 8-4 所示。

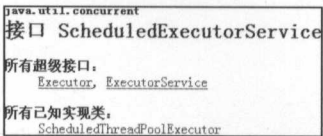


图 8-2 接口的 ScheduledExecutor-
Service 基本信息

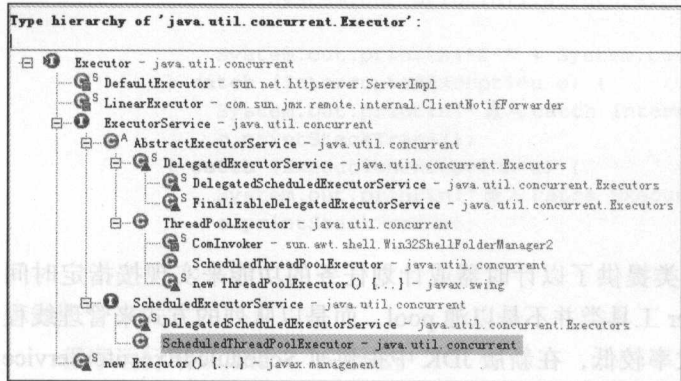


图 8-3 接口 Executor 继承及实现关系

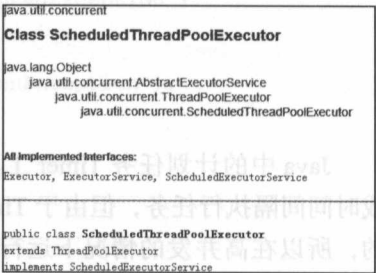


图 8-4 类 ScheduledThreadPoolExecutor
继承关系

类 ScheduledThreadPoolExecutor 的 API 结构如图 8-5 所示。

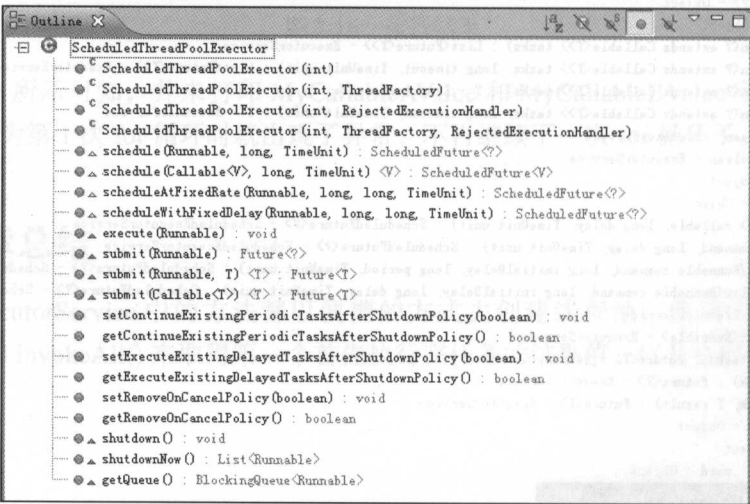


图 8-5 类 ScheduledThreadPoolExecutor 的 API 列表

从图 8-5 中的方法列表来看, 部分方法是父类 `ThreadPoolExecutor` 提供并在子类 `ScheduledThreadPoolExecutor` 中重写的, 比如 `submit()` 重载方法或 `shutdown()` 等方法。

8.2 ScheduledThreadPoolExecutor 使用 Callable 延迟运行

本示例使用 `Callable` 接口进行任务延迟运行的实验, 具有返回值的功能。

创建测试用的项目 `ScheduledThreadPoolExecutor_1`, 类 `MyCallableA.java` 和 `MyCallableB.java` 代码如下:

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {
    @Override
    public String call() throws Exception {
        try {
            System.out.println("callA begin "
                + Thread.currentThread().getName() + " "
                + System.currentTimeMillis());
            Thread.sleep(3000);
            System.out.println("callA end "
                + Thread.currentThread().getName() + " "
                + System.currentTimeMillis());
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "returnA";
    }
}
```

```
package mycallable;

import java.util.concurrent.Callable;

public class MyCallableB implements Callable<String> {
    @Override
    public String call() throws Exception {
        System.out.println("callB begin " + Thread.currentThread().getName()
            + " " + System.currentTimeMillis());
        System.out.println("callB end " + Thread.currentThread().getName()
            + " " + System.currentTimeMillis());
        return "returnB";
    }
}
```

类 `Run1.java` 代码如下:

```
package test.run;
```

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run1 {
    public static void main(String[] args) {
        try {
            List<Callable> callableList = new ArrayList();
            callableList.add(new MyCallableA());
            callableList.add(new MyCallableB());
            // 调用方法 newSingleThreadScheduledExecutor()
            // 取得一个单任务的计划任务执行池
            ScheduledExecutorService executor = Executors
                .newSingleThreadScheduledExecutor();
            ScheduledFuture<String> futureA = executor.schedule(callableList
                .get(0), 4L, TimeUnit.SECONDS);
            ScheduledFuture<String> futureB = executor.schedule(callableList
                .get(1), 4L, TimeUnit.SECONDS);
            System.out.println("      X=" + System.currentTimeMillis());
            System.out.println(" 返回值 A: " + futureA.get());
            System.out.println(" 返回值 B: " + futureB.get());
            System.out.println("      Y=" + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 8-6 所示。

从 X 到 Y 的运行时间为 7 秒，阻塞点是 get() 方法。

在运行结果中可以发现：

```

X=1435367513671
callA begin pool-1-thread-1 1435367517671
callA end pool-1-thread-1 1435367520671
返回值A: returnA
callB begin pool-1-thread-1 1435367520671
callB end pool-1-thread-1 1435367520671
返回值B: returnB
Y=1435367520671

```

图 8-6 按顺序延迟运行成功

`public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)`

方法中的第 2 个参数在多个任务中同时消耗时间，并不是一个任务执行完毕后再等待 4 秒继续执行的效果。由于第 1 个任务从计划任务到运行结束需要用时 7 秒，那么第 2 个任务其实是在第 4 秒被执行，由于是单任务的计划任务池，所以第 2 个任务的执行时间被延后 3 秒。

在此实验中使用工厂类 `Executors` 的 `newSingleThreadScheduledExecutor()` 方法来创建 `ScheduledExecutorService` 对象，但返回的真正对象却是 `ScheduledThreadPoolExecutor`，因为 `ScheduledThreadPoolExecutor` 实现了 `ScheduledExecutorService` 接口。

方法 `newSingleThreadScheduledExecutor()` 在 JDK 中的源代码如下:

```
public static ScheduledExecutorService newSingleThreadScheduledExecutor() {
    return new DelegatedScheduledExecutorService
        (new ScheduledThreadPoolExecutor(1));
}
```

从源代码中还可以发现,调用 `newSingleThreadScheduledExecutor()` 方法时,在源代码中实例化 `new ScheduledThreadPoolExecutor(1)` 对象时传入的参数为 1,是单任务执行的计划任务池。

前面的示例使用 `newSingleThreadScheduledExecutor()` 方法创建的是单任务的计划任务池,那使用 `newScheduledThreadPool(poolSize)` 方法呢?

创建类 `Run2.java` 代码如下:

```
package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallableA;
import mycallable.MyCallableB;

public class Run2 {
    public static void main(String[] args) {
        try {
            List<Callable> callableList = new ArrayList();
            callableList.add(new MyCallableA());
            callableList.add(new MyCallableB());
            // 调用方法 newScheduledThreadPool(corePoolSize >1)
            // 取得一个同时运行 corePoolSize 任务个数的计划任务执行池 ScheduledExecutorService
            executor = Executors
                .newScheduledThreadPool(2);
            ScheduledFuture<String> futureA = executor.schedule(callableList
                .get(0), 4L, TimeUnit.SECONDS);
            ScheduledFuture<String> futureB = executor.schedule(callableList
                .get(1), 4L, TimeUnit.SECONDS);
            System.out.println("X=" + System.currentTimeMillis());
            System.out.println("返回值 A: " + futureA.get());
            System.out.println("返回值 B: " + futureB.get());
            System.out.println("Y=" + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
}
}

```

程序运行效果如图 8-7 所示。

注意：如果使用 `newScheduledThreadPool(1)` 的写法创建出来的任务池还是单任务的。

方法 `newScheduledThreadPool(poolSize)` 在 JDK 中的源代码如下：

```

X=1435367563390
callA begin pool-1-thread-1 1435367567390
callB begin pool-1-thread-2 1435367567390
callB end pool-1-thread-2 1435367567390
callA end pool-1-thread-1 1435367570390
返回值A: returnA
返回值B: returnB
Y=1435367570390

```

图 8-7 两个任务异步同时运行

```

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

```

8.3 ScheduledThreadPoolExecutor 使用 Runnable 延迟运行

本示例使用 `Runnable` 接口进行无返回值的计划任务实验。

创建测试用的项目 `ScheduledThreadPoolExecutor_2`，类 `MyRunnableA.java` 和 `MyRunnableB.java` 代码如下：

```
package mycallable;
```

```
public class MyRunnableA implements Runnable {
```

```
    @Override
```

```
    public void run() {
```

```
        try {
```

```
            System.out.println("runnableA begin " +
```

```
                Thread.currentThread().getName() + " " +
```

```
                System.currentTimeMillis());
```

```
            Thread.sleep(3000);
```

```
            System.out.println("runnableA end " +
```

```
                Thread.currentThread().getName() + " " +
```

```
                System.currentTimeMillis());
```

```
        } catch (Exception e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
}
```

```
package mycallable;
```

```
public class MyRunnableB implements Runnable {
```

```
    @Override
```

```
    public void run() {
```

```
        System.out.println("runnableB begin " + Thread.currentThread().getName()
            + " " + System.currentTimeMillis());
```

```
        System.out.println("runnableB end " + Thread.currentThread().getName())
```

```

        + " " + System.currentTimeMillis());
    }
}

```

类 Run.java 代码如下:

```

package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnableA;
import mycallable.MyRunnableB;

public class Run {
    public static void main(String[] args) {
        List<Runnable> runnableList = new ArrayList();
        runnableList.add(new MyRunnableA());
        runnableList.add(new MyRunnableB());
        ScheduledExecutorService executor = Executors
            .newSingleThreadScheduledExecutor();

        System.out.println("    X=" + System.currentTimeMillis());
        executor.schedule(runnableList.get(0), 0L, TimeUnit.SECONDS);
        executor.schedule(runnableList.get(1), 0L, TimeUnit.SECONDS);
        System.out.println("    Y=" + System.currentTimeMillis());

    }
}

```

程序运行结果如图 8-8 所示。

```

X=1435367862406
Y=1435367862406
runnableA begin pool-1-thread-1 1435367862406
runnableA end pool-1-thread-1 1435367865406
runnableB begin pool-1-thread-1 1435367865406
runnableB end pool-1-thread-1 1435367865406

```

8.4 延迟运行并取得返回值

图 8-8 无返回运行效果成功

创建测试用的项目 ScheduledThreadPoolExecutor_3, 类 MyCallableA.java 和 MyCallableB.java 代码如下:

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallableA implements Callable<String> {
    public String call() throws Exception {
        System.out.println("a call run =" + System.currentTimeMillis());
        return "returnA";
    }
}

```


类 Run.java 代码如下：

```
package test.run;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

import mycallable.MyCallableA;

public class Run {
    public static void main(String[] args) {
        try {
            List<Callable> callableList = new ArrayList();
            callableList.add(new MyCallableA());
            ScheduledExecutorService executor = Executors
                .newSingleThreadScheduledExecutor();
            System.out.println("X=" + System.currentTimeMillis());
            ScheduledFuture<String> futureA = executor.schedule(callableList
                .get(0), 4L, TimeUnit.SECONDS);
            System.out.println(futureA.get() + "A="
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 8-9 所示。

延迟 4 秒运行并成功取得了返回值。

Run (1) [Java Application] C:\jdk1.7
X=1435368083937
a call run =1435368087937
returnA A=1435368087937

图 8-9 获取返回值成功

8.5 使用 scheduleAtFixedRate() 方法实现周期性执行

此示例测试的是执行任务时间大于 > period 预定的周期时间，也就是产生了超时的效果。

创建测试用的项目 ScheduledThreadPoolExecutor_4，类 MyRunnable.java 代码如下：

```
package mycallable;

public class MyRunnable implements Runnable {

    @Override
    public void run() {
```

```

try {
    System.out.println("        begin =" + System.currentTimeMillis()
        + " ThreadName=" + Thread.currentThread().getName());
    Thread.sleep(4000);
    System.out.println("        end =" + System.currentTimeMillis()
        + " ThreadName=" + Thread.currentThread().getName());
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

```

类 Run.java 代码如下:

```

package test.run;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run {
    public static void main(String[] args) {
        ScheduledExecutorService executor = Executors
            .newSingleThreadScheduledExecutor();
        System.out.println("        X=" + System.currentTimeMillis());
        executor.scheduleAtFixedRate(new MyRunnable(), 1, 2, TimeUnit.SECONDS);
        System.out.println("        Y=" + System.currentTimeMillis());
    }
}

```

程序运行结果如图 8-10 所示。

下面继续实验,测试的是执行任务时间小于 <period 的时间。

创建测试用的项目 ScheduledThreadPoolExecutor_4_1,
类 MyRunnable.java 代码如下:

```

package mycallable;

public class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("        begin =" + System.currentTimeMillis()
            + " ThreadName=" + Thread.currentThread().getName());
        System.out.println("        end =" + System.currentTimeMillis()
            + " ThreadName=" + Thread.currentThread().getName());
    }
}

```

类 Run.java 代码如下:

```

X=1420859043843
Y=1420859043843
begin =1420859044843 ThreadName=pool-1-thread-1
end =1420859048843 ThreadName=pool-1-thread-1
begin =1420859048843 ThreadName=pool-1-thread-1
end =1420859052843 ThreadName=pool-1-thread-1
begin =1420859052843 ThreadName=pool-1-thread-1
end =1420859056843 ThreadName=pool-1-thread-1
begin =1420859056843 ThreadName=pool-1-thread-1
end =1420859060843 ThreadName=pool-1-thread-1
begin =1420859060843 ThreadName=pool-1-thread-1
end =1420859064843 ThreadName=pool-1-thread-1
begin =1420859064843 ThreadName=pool-1-thread-1
end =1420859068843 ThreadName=pool-1-thread-1
begin =1420859068843 ThreadName=pool-1-thread-1

```

图 8-10 成功周期性执行

```

package test.run;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run {
    public static void main(String[] args) {
        ScheduledExecutorService executor = Executors
            .newSingleThreadScheduledExecutor();
        System.out.println("X=" + System.currentTimeMillis());
        executor.scheduleAtFixedRate(new MyRunnable(), 1, 2, TimeUnit.SECONDS);
        System.out.println("Y=" + System.currentTimeMillis());
    }
}

```

程序运行结果如图 8-11 所示。

注意，`scheduleAtFixedRate()` 方法返回的 `ScheduledFuture` 对象无法获得返回值，也就是 `scheduleAtFixedRate()` 方法不具有获得返回值的功能，而 `schedule()` 方法却可以获得返回值。所以当使用 `scheduleAtFixedRate()` 方法实现重复运行任务的效果时，需要结合自定义 `Runnable` 接口的实现类，不要使用 `FutureTask` 类，因为 `FutureTask` 类并不能实现重复运行的效果。

```

X=1420859197140
Y=1420859197140
begin =1420859198140 ThreadName=pool-1-thread-1
end =1420859198140 ThreadName=pool-1-thread-1
begin =1420859200140 ThreadName=pool-1-thread-1
end =1420859200140 ThreadName=pool-1-thread-1
begin =1420859202140 ThreadName=pool-1-thread-1
end =1420859202140 ThreadName=pool-1-thread-1
begin =1420859204140 ThreadName=pool-1-thread-1
end =1420859204140 ThreadName=pool-1-thread-1
begin =1420859206140 ThreadName=pool-1-thread-1
end =1420859206140 ThreadName=pool-1-thread-1

```

图 8-11 成功周期性执行

8.6 使用 `scheduleWithFixedDelay()` 方法实现周期性执行

方法 `scheduleWithFixedDelay()` 的主要作用是设置多个任务之间固定的运行时间间隔。此示例测试的是执行任务时间大于 `> period` 预定的时间。

创建测试用的项目 `ScheduledThreadPoolExecutor_5`，类 `MyRunnable.java` 代码如下：

```

package mycallable;

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        try {
            System.out.println("begin =" + System.currentTimeMillis()
                + " ThreadName=" + Thread.currentThread().getName());
            Thread.sleep(4000);
            System.out.println("end =" + System.currentTimeMillis()
                + " ThreadName=" + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

类 Run.java 代码如下:

```
package test.run;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run {
    public static void main(String[] args) {
        ScheduledExecutorService executor = Executors
            .newSingleThreadScheduledExecutor();
        System.out.println("      X=" + System.currentTimeMillis());
        executor.scheduleWithFixedDelay(new MyRunnable(), 1, 2,
            TimeUnit.SECONDS);
        System.out.println("      Y=" + System.currentTimeMillis());
    }
}
```

程序运行结果如图 8-12 所示。

继续实验, 下面测试的是执行任务时间小于 <period 的时间。

创建测试用的项目 ScheduledThreadPoolExecutor_5_1,

类 MyRunnable.java 代码如下:

```
package mycallable;

public class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("      begin =" + System.currentTimeMillis()
            + " ThreadName=" + Thread.currentThread().getName());
        System.out.println("      end =" + System.currentTimeMillis()
            + " ThreadName=" + Thread.currentThread().getName());
    }
}
```

类 Run.java 代码如下:

```
package test.run;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
```

```
X=1420859584968
Y=1420859584968
begin =1420859585968 ThreadName=pool-1-thread-1
end =1420859589968 ThreadName=pool-1-thread-1
begin =1420859591968 ThreadName=pool-1-thread-1
end =1420859595968 ThreadName=pool-1-thread-1
begin =1420859597968 ThreadName=pool-1-thread-1
end =1420859601968 ThreadName=pool-1-thread-1
begin =1420859603968 ThreadName=pool-1-thread-1
end =1420859607968 ThreadName=pool-1-thread-1
begin =1420859609968 ThreadName=pool-1-thread-1
end =1420859613968 ThreadName=pool-1-thread-1
```

图 8-12 成功周期性执行

```

import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run {
    public static void main(String[] args) {
        ScheduledExecutorService executor = Executors
            .newSingleThreadScheduledExecutor();
        System.out.println("        X=" + System.currentTimeMillis());
        executor.scheduleWithFixedDelay(new MyRunnable(), 1, 2,
            TimeUnit.SECONDS);
        System.out.println("        Y=" + System.currentTimeMillis());
    }
}

```

程序运行结果如图 8-13 所示。

通过本节两个案例的运行结果可知，方法 `scheduleWithFixedDelay()` 并没有超时与非超时的情况，参数 `long delay` 的主要作用就是下一个任务的开始时间与上一个任务的结束时间的的时间间隔。

```

X=1420859697515
Y=1420859697515
begin =1420859698515 ThreadName=pool-1-thread-1
end   =1420859698515 ThreadName=pool-1-thread-1
begin =1420859700515 ThreadName=pool-1-thread-1
end   =1420859700515 ThreadName=pool-1-thread-1
begin =1420859702515 ThreadName=pool-1-thread-1
end   =1420859702515 ThreadName=pool-1-thread-1
begin =1420859704515 ThreadName=pool-1-thread-1
end   =1420859704515 ThreadName=pool-1-thread-1
begin =1420859706515 ThreadName=pool-1-thread-1
end   =1420859706515 ThreadName=pool-1-thread-1
begin =1420859708515 ThreadName=pool-1-thread-1
end   =1420859708515 ThreadName=pool-1-thread-1

```

图 8-13 成功周期性执行

8.7 使用 `getQueue()` 与 `remove()` 方法

方法 `getQueue()` 的作用是取得队列中的任务，而这些任务是未来将要运行的，正在运行的任务不在此队列中。使用 `scheduleAtFixedRate()` 和 `scheduleWithFixedDelay()` 两个方法实现周期性执行任务时，未来欲执行的任务都是放入此队列中。

创建测试用的项目 `ScheduledThreadPoolExecutor_6`，类 `MyRunnable.java` 代码如下：

```

package mycallable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}

```

```

@Override
public void run() {
    System.out.println("run! username=" + username + " "
        + Thread.currentThread().getName());
}
}

```

类 Run1.java 代码如下:

```

package test.run;

import java.util.Iterator;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run1 {
    public static void main(String[] args) {

        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            10);

        Runnable runnable1 = new MyRunnable("A");
        Runnable runnable2 = new MyRunnable("B");
        Runnable runnable3 = new MyRunnable("C");
        Runnable runnable4 = new MyRunnable("D");
        Runnable runnable5 = new MyRunnable("E");

        System.out.println(runnable1.hashCode());
        System.out.println(runnable2.hashCode());
        System.out.println(runnable3.hashCode());
        System.out.println(runnable4.hashCode());
        System.out.println(runnable5.hashCode());

        executor.scheduleAtFixedRate(runnable1, 10, 2, TimeUnit.SECONDS);
        executor.scheduleAtFixedRate(runnable2, 10, 2, TimeUnit.SECONDS);
        executor.scheduleAtFixedRate(runnable3, 10, 2, TimeUnit.SECONDS);
        executor.scheduleAtFixedRate(runnable4, 10, 2, TimeUnit.SECONDS);
        executor.scheduleAtFixedRate(runnable5, 10, 2, TimeUnit.SECONDS);

        System.out.println("");

        BlockingQueue<Runnable> queue = executor.getQueue();
        Iterator<Runnable> iterator = queue.iterator();
        while (iterator.hasNext()) {
            Runnable runnable = (Runnable) iterator.next();
            System.out.println(" 队列中的: " + runnable);
        }
    }
}

```


程序运行结果如图 8-14 所示。

```
18433730
7634850
6783657
28524038
24791433

队列中的: java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask$804ce7a
队列中的: java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask$10fd7f6
队列中的: java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask$12b6c09
队列中的: java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask$1e2b6fa
队列中的: java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask$1682598
run! username=C pool-1-thread-2
run! username=D pool-1-thread-4
run! username=B pool-1-thread-3
run! username=A pool-1-thread-1
run! username=E pool-1-thread-5
run! username=A pool-1-thread-6
run! username=B pool-1-thread-2
run! username=D pool-1-thread-3
run! username=C pool-1-thread-6
run! username=E pool-1-thread-8
```

图 8-14 队列成功打印

类 Run2.java 代码如下：

```
package test.run;

import java.util.Iterator;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run2 {
    public static void main(String[] args) throws InterruptedException {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            1);

        Runnable runnable1 = new MyRunnable("A");
        Runnable runnable2 = new MyRunnable("B");

        ScheduledFuture future1 = executor.scheduleAtFixedRate(runnable1, 0, 2,
            TimeUnit.SECONDS);
        Thread.sleep(1000);
        ScheduledFuture future2 = executor.scheduleAtFixedRate(runnable2, 10,
            2, TimeUnit.SECONDS);
        Thread.sleep(5000);
        // 注意: remove() 方法的参数是 ScheduledFuture 数据类型
        System.out.println(executor.remove((Runnable) future2));
        System.out.println("");

        BlockingQueue<Runnable> queue = executor.getQueue();
        Iterator<Runnable> iterator = queue.iterator();
        while (iterator.hasNext()) {
            Runnable runnable = (Runnable) iterator.next();
```

```

        System.out.println(" 队列中的: " + runnable);
    }
}
}

```

程序运行结果如图 8-15 所示。

```

run! username=A pool-1-thread-1
run! username=A pool-1-thread-1
run! username=A pool-1-thread-1
true
队列中的: java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask@c943d1
run! username=A pool-1-thread-1
run! username=A pool-1-thread-1
run! username=A pool-1-thread-1
run! username=A pool-1-thread-1
run! username=A pool-1-thread-1
run! username=A pool-1-thread-1

```

图 8-15 成功删除队列中的任务

8.8 方法 setExecuteExistingDelayedTasksAfterShutdownPolicy() 的使用

方法 setExecuteExistingDelayedTasksAfterShutdownPolicy() 的作用是当对 ScheduledThreadPoolExecutor 执行了 shutdown() 方法时,任务是否继续运行,默认值是 true,也就是当调用了 shutdown() 方法时任务还是继续运行,当使用 setExecuteExistingDelayedTasksAfterShutdownPolicy(false) 时任务不再运行。

创建测试用的项目 ScheduledThreadPoolExecutor_7,类 MyRunnable.java 代码如下:

```

package mycallable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public void run() {

```



```

        System.out.println("run! username=" + username + " "
            + Thread.currentThread().getName() + " "
            + System.currentTimeMillis());
    }
}

```

类 Run1.java 代码如下：

```

package test.run;

import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run1 {
    public static void main(String[] args) throws InterruptedException {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            1);
        Runnable runnable1 = new MyRunnable("A");
        Runnable runnable2 = new MyRunnable("B");
        executor.schedule(runnable1, 3, TimeUnit.SECONDS);
        executor.shutdown();
        System.out.println("已经 shutdown 了");
    }
}

```

程序运行结果如图 8-16 所示。

类 Run2.java 代码如下：

```

package test.run;

import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run2 {
    public static void main(String[] args) {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            1);
        Runnable runnable1 = new MyRunnable("A");
        Runnable runnable2 = new MyRunnable("B");
        executor.schedule(runnable1, 3, TimeUnit.SECONDS);
        executor.setExecuteExistingDelayedTasksAfterShutdownPolicy(false);
        executor.shutdown();
        System.out.println("已经 shutdown 了");
    }
}

```

程序运行结果如图 8-17 所示。

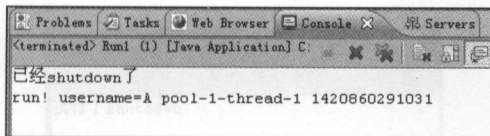


图 8-16 虽然 shutdown 但任务还是执行了

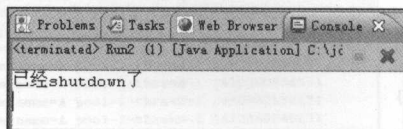


图 8-17 任务被取消执行了进程销毁了

方法 `setExecuteExistingDelayedTasksAfterShutdownPolicy()` 可以与 `schedule()` 和 `shutdown()` 方法联合使用, 但 `setExecuteExistingDelayedTasksAfterShutdownPolicy()` 方法不能与 `scheduleAtFixedRate()` 和 `scheduleWithFixedDelay()` 方法联合使用。那么如果要想实现 shutdown 关闭线程池后, 池中的任务还会继续重复运行, 则要将 `scheduleAtFixedRate()` 和 `scheduleWithFixedDelay()` 方法与 `setContinueExistingPeriodicTasksAfterShutdownPolicy()` 方法联合使用。

8.9 方法 `setContinueExistingPeriodicTasksAfterShutdownPolicy()`

方法 `setContinueExistingPeriodicTasksAfterShutdownPolicy()` 传入 `true` 的作用是当使用 `scheduleAtFixedRate()` 方法或 `scheduleWithFixedDelay()` 方法时, 如果调用 `ScheduledThreadPoolExecutor` 对象的 `shutdown()` 方法, 任务还会继续运行, 传入 `false` 时任务不运行, 进程销毁。

创建测试用的项目 `ScheduledThreadPoolExecutor_8`, 类 `MyRunnable.java` 代码如下:

```
package mycallable;

public class MyRunnable implements Runnable {

    private String username;

    public MyRunnable(String username) {
        super();
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public void run() {
        System.out.println("run! username=" + username + " "
            + Thread.currentThread().getName() + " ");
    }
}
```

```

        + System.currentTimeMillis());
    }
}

```

类 Run1.java 代码如下：

```

package test.run;

import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import mycallable.MyRunnable;

public class Run1 {
    public static void main(String[] args) {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            10);
        Runnable runnable1 = new MyRunnable("A");
        executor.scheduleAtFixedRate(runnable1, 1, 2, TimeUnit.SECONDS);
        executor.shutdown();
        System.out.println("执行了 shutdown!");
    }
}

```

程序运行结果如图 8-18 所示。

如果使用 `scheduleAtFixedRate()` 结合 `shutdown()` 方法想实现任务继续运行的效果，则必须使用 `setContinueExistingPeriodicTasksAfterShutdownPolicy(true)` 代码。

类 Run2.java 代码如下：

```

package test.run;

import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import mycallable.MyRunnable;

public class Run2 {
    public static void main(String[] args) {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            10);
        Runnable runnable1 = new MyRunnable("A");
        executor.scheduleAtFixedRate(runnable1, 1, 2, TimeUnit.SECONDS);
        executor.setContinueExistingPeriodicTasksAfterShutdownPolicy(true);
        executor.shutdown();
        System.out.println("执行了 shutdown!");
    }
}

```

程序运行结果如图 8-19 所示。

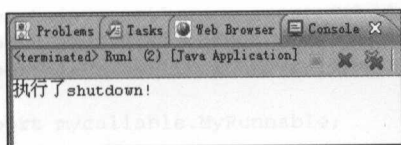


图 8-18 执行 shutdown 后进程销毁了

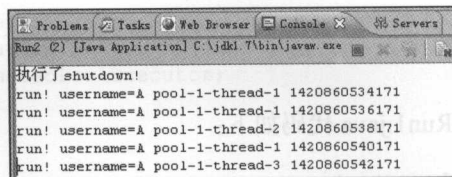


图 8-19 继续执行

8.10 使用 cancel(boolean) 与 setRemoveOnCancelPolicy() 方法

方法 cancel(boolean) 的作用设定是否取消任务。

方法 setRemoveOnCancelPolicy(boolean) 的作用设定是否将取消后的任务从队列中清除。

创建测试用的项目 ScheduledThreadPoolExecutor_9, 类 MyRunnable.java 代码如下:

```
package mycallable;
```

```
public class MyRunnable implements Runnable {
```

```
    private String username;
```

```
    public MyRunnable(String username) {
```

```
        super();
```

```
        this.username = username;
```

```
    }
```

```
    public String getUsername() {
```

```
        return username;
```

```
    }
```

```
    public void setUsername(String username) {
```

```
        this.username = username;
```

```
    }
```

```
@Override
```

```
public void run() {
```

```
    try {
```

```
        while (true) {
```

```
            if (Thread.currentThread().isInterrupted() == true) {
```

```
                throw new InterruptedException();
```

```
            }
```

```
            System.out.println("run! username=" + username + " "
```

```
                + Thread.currentThread().getName());
```

```
            Thread.sleep(1000);
```

```
        }
```

```
    } catch (InterruptedException e) {
```

```
        e.printStackTrace();
```

```
        System.out.println("中断了! ");
```

```
    }
```


类 Run1.java 代码如下：

```
package test.run;

import java.util.Iterator;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run1 {
    public static void main(String[] args) {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            10);
        Runnable runnable1 = new MyRunnable("A");
        ScheduledFuture future = executor.schedule(runnable1, 1,
            TimeUnit.SECONDS);
        System.out.println(future.cancel(true));
        System.out.println("");
        BlockingQueue<Runnable> queue = executor.getQueue();
        Iterator<Runnable> iterator = queue.iterator();
        while (iterator.hasNext()) {
            Runnable runnable = (Runnable) iterator.next();
            System.out.println("队列中的: " + runnable);
        }
        System.out.println("main end!");
    }
}
```

代码 `executor.schedule(runnable1, 1, TimeUnit.SECONDS);` 中的第 2 个参数为 1，说明要把任务放入队列中，1 秒之后再运行。

程序运行结果如图 8-20 所示。

```
true
队列中的: java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask@18052d8b
main end!
```

图 8-20 任务成功被取消永远不会被执行

此运行结果说明，当执行 `cancel()` 方法后任务虽然被成功取消，但还是在队列中存在。此结果也说明了，在队列中的任务被成功取消，任务也不再运行。

上面的示例说明队列中有任务，下面的示例代码则说明队列中没有任务的情况，创建类 `Run1_ext.java` 代码如下：

```
package test.run;
```

```

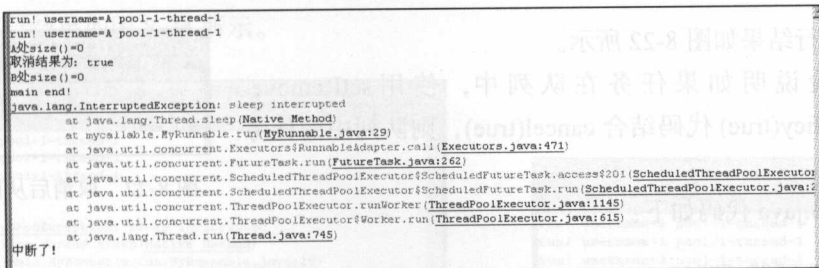
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run1_ext {
    public static void main(String[] args) throws InterruptedException {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            10);
        Runnable runnable1 = new MyRunnable("A");
        // 第2个参数为0则任务不放入队列中马上运行
        ScheduledFuture future = executor.schedule(runnable1, 0,
            TimeUnit.SECONDS);
        Thread.sleep(2000);
        BlockingQueue<Runnable> queue = executor.getQueue();
        System.out.println("A处 size()=" + queue.size());
        System.out.println("取消结果为: " + future.cancel(true));
        queue = executor.getQueue();
        System.out.println("B处 size()=" + queue.size());
        System.out.println("main end!");
    }
}

```

程序运行结果如图 8-21 所示。



```

run! username=A pool-1-thread-1
run! username=A pool-1-thread-1
A处size()=0
取消结果为: true
B处size()=0
main end!
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at mycallable.MyRunnable.run(MyRunnable.java:29)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:1145)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:615)
    at java.util.concurrent.ThreadPoolExecutor.run(ThreadPoolExecutor.java:745)
中断了!

```

图 8-21 任务队列中无任务 size 为 0

队列中无任务的原因是任务马上被执行了，并未放入队列中，所以队列的 size() 为 0。此实验也说明正在运行中的任务执行 cancel() 方法结合 if (Thread.currentThread().isInterrupted() == true) 判断时，是可以让任务结束。

总结如下：

- 1) 在队列中的任务可以取消，任务也不再运行。
- 2) 正在运行的任务可以停止，但要结合 if (Thread.currentThread().isInterrupted() == true) 判断。

类 Run2.java 代码如下：

```

package test.run;

import java.util.Iterator;

```

```

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run2 {
    public static void main(String[] args) throws InterruptedException {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            10);
        Runnable runnable1 = new MyRunnable("A");
        ScheduledFuture future = executor.schedule(runnable1, 1,
            TimeUnit.SECONDS);
        executor.setRemoveOnCancelPolicy(true);
        System.out.println(future.cancel(true));
        System.out.println("");
        BlockingQueue<Runnable> queue = executor.getQueue();
        Iterator<Runnable> iterator = queue.iterator();
        while (iterator.hasNext()) {
            Runnable runnable = (Runnable) iterator.next();
            System.out.println(" 队列中的: " + runnable);
        }
        System.out.println("main end!");
    }
}

```

程序运行结果如图 8-22 所示。

此实验说明如果任务在队列中，使用 `setRemoveOnCancelPolicy(true)` 代码结合 `cancel(true)`，则队列中的任务被删除。

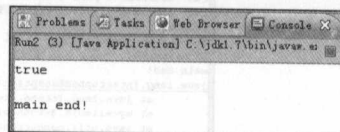


图 8-22 取消后从队列中删除

类 Run3.java 代码如下：

```

package test.run;

import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run3 {
    public static void main(String[] args) throws InterruptedException {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            10);
        Runnable runnable1 = new MyRunnable("A");
        ScheduledFuture future = executor.schedule(runnable1, 0,
            TimeUnit.SECONDS);
        Thread.sleep(3000);
        System.out.println(future.cancel(true));
        System.out.println("main end!");
    }
}

```

```

    }
}

```

程序运行结果如图 8-23 所示。

类 Run4.java 代码如下：

```

package test.run;

import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import mycallable.MyRunnable;

public class Run4 {
    public static void main(String[] args) throws InterruptedException {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
            10);
        Runnable runnable1 = new MyRunnable("A");
        ScheduledFuture future = executor.schedule(runnable1, 0,
            TimeUnit.SECONDS);
        Thread.sleep(3000);
        System.out.println(future.cancel(false));
        System.out.println("main end!");
    }
}

```

程序运行结果如图 8-24 所示。

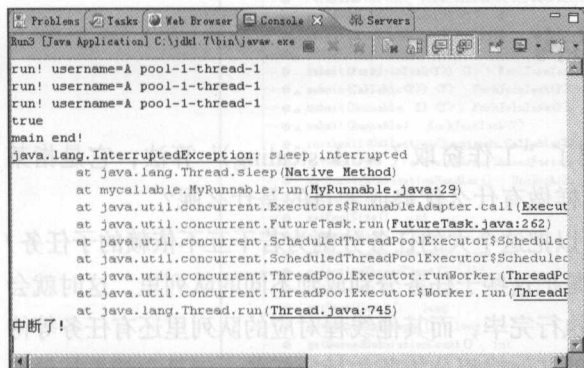


图 8-23 参数 true 使运行中的任务可以中断

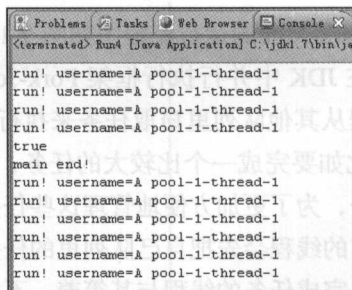


图 8-24 参数 false 使运行中的任务继续执行

8.11 本章总结

本章介绍了基于线程池 `ThreadPoolExecutor` 的 `ScheduledThreadPoolExecutor` 计划任务执行池对象，使用此类可以高效地实现计划任务线程池，不再重复创建 `Thread` 对象，提高了运行效率。此类也支持间隔运行的功能。

Fork-Join 分治编程

在 JDK1.7 版本中提供了 Fork/Join 并行执行任务框架，它的主要作用是把大任务分割成若干个小任务，再对每个小任务得到的结果进行汇总，此种开发方法也叫分治编程，分治编程可以极大地利用 CPU 资源，提高任务执行的效率，也是目前与多线程有关的前沿技术。

9.1 Fork-Join 分治编程与类结构

Fork-Join 的运行流程图如图 9-1 所示。

在 JDK 中并行执行框架 Fork-Join 使用了“工作窃取（work-stealing）”算法，它是指某个线程从其他队列里窃取任务来执行，那这样做有什么优势或者目的是什么呢？

比如要完成一个比较大的任务，完全可以把这个大的任务分割为若干互不依赖的子任务 / 小任务，为了更加方便地管理这些任务，于是把这些子任务分别放到不同的队列里，这时就会出现有的线程会先把自己队列里的任务快速执行完毕，而其他线程对应的队列里还有任务等待处理，完成任务的线程与其等着，不如去帮助其他线程分担要执行的任务，于是它就去其他线程的队列里窃取一个任务来执行，这就是所谓的“工作窃取（work-stealing）”算法。

在 JDK1.7 中实现分治编程需要使用 ForkJoinPool 类，此类的主要作用是创建一个任务池，类信息如图 9-2 所示。

类代码如下：

```
public class ForkJoinPool extends AbstractExecutorService {
```

该类也是从 AbstractExecutorService 类继承下来的。

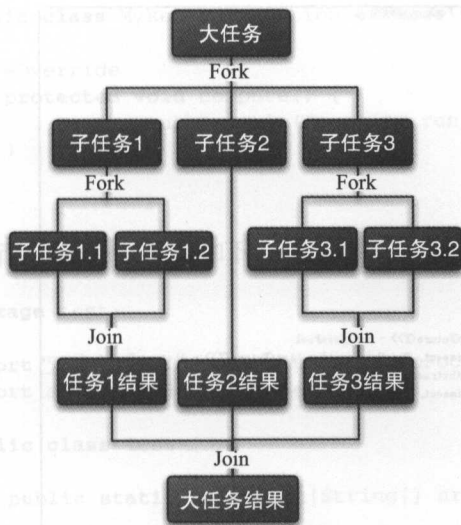


图 9-1 框架 Fork-Join 运行的流程

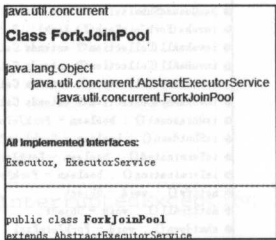


图 9-2 类结构 ForkJoinPool

类 ForkJoinPool 方法声明如图 9-3 所示。

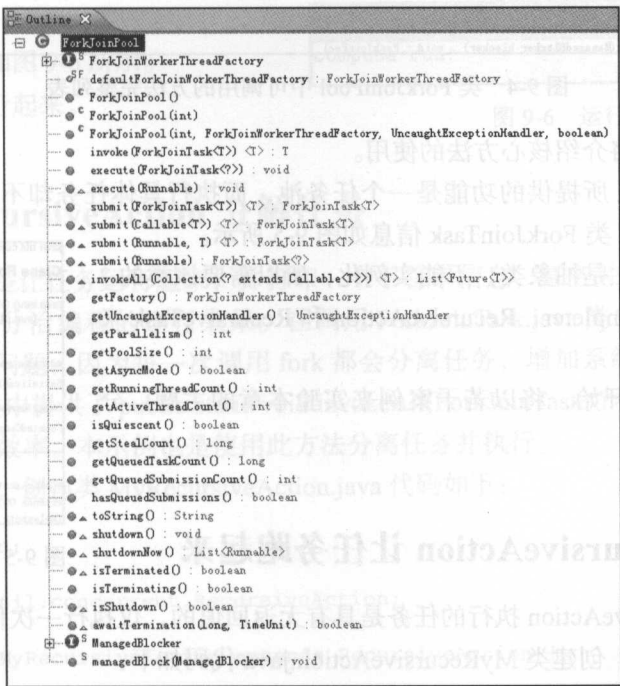


图 9-3 类 ForkJoinPool 方法声明

完整的可调用方法列表如图 9-4 所示。


```

    @SuppressWarnings("unchecked")
    public void awaitTermination(long timeout, TimeUnit unit) throws InterruptedException {
        boolean = ForkJoinPool
    }
    public boolean equals(Object obj) {
        boolean = Object
    }
    public void execute(ForkJoinTask<V> task) {
        void = ForkJoinPool
    }
    public void execute(Runnable task) {
        void = ForkJoinPool
    }
    public int getActiveThreadCount() {
        int = ForkJoinPool
    }
    public boolean getAsyncMode() {
        boolean = ForkJoinPool
    }
    public Class<V> getClass() {
        Class<V> = Object
    }
    public ForkJoinWorkerThreadFactory getFactory() {
        ForkJoinWorkerThreadFactory = ForkJoinPool
    }
    public int getParallelism() {
        int = ForkJoinPool
    }
    public int getPoolSize() {
        int = ForkJoinPool
    }
    public int getQueuedSubmissionCount() {
        int = ForkJoinPool
    }
    public long getQueuedTaskCount() {
        long = ForkJoinPool
    }
    public int getRunningThreadCount() {
        int = ForkJoinPool
    }
    public long getStealCount() {
        long = ForkJoinPool
    }
    public UncaughtExceptionHandler getUncaughtExceptionHandler() {
        UncaughtExceptionHandler = ForkJoinPool
    }
    public int hashCode() {
        int = Object
    }
    public boolean hasQueuedSubmissions() {
        boolean = ForkJoinPool
    }
    public T invoke(ForkJoinTask<T> task) {
        T = ForkJoinPool
    }
    public List<Future<T>> invokeAll(Collection<Callable<T>> tasks) {
        List<Future<T>> = ForkJoinPool
    }
    public List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit) {
        List<Future<T>> = AbstractExecutorService
    }
    public T invokeAny(Collection<Callable<T>> tasks) {
        T = AbstractExecutorService
    }
    public T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit) {
        T = AbstractExecutorService
    }
    public boolean isQuiescent() {
        boolean = ForkJoinPool
    }
    public boolean isShutdown() {
        boolean = ForkJoinPool
    }
    public boolean isTerminated() {
        boolean = ForkJoinPool
    }
    public boolean isTerminating() {
        boolean = ForkJoinPool
    }
    public void notify() {
        void = Object
    }
    public void notifyAll() {
        void = Object
    }
    public void shutdown() {
        void = ForkJoinPool
    }
    public List<Runnable> shutdownNow() {
        List<Runnable> = ForkJoinPool
    }
    public ForkJoinTask<V> submit(Callable<V> task) {
        ForkJoinTask<V> = ForkJoinPool
    }
    public ForkJoinTask<V> submit(ForkJoinTask<V> task) {
        ForkJoinTask<V> = ForkJoinPool
    }
    public ForkJoinTask<V> submit(Runnable task) {
        ForkJoinTask<V> = ForkJoinPool
    }
    public ForkJoinTask<V> submit(Runnable task, T result) {
        ForkJoinTask<V> = ForkJoinPool
    }
    public String toString() {
        String = ForkJoinPool
    }
    public void wait() {
        void = Object
    }
    public void wait(long timeout) {
        void = Object
    }
    public void wait(long timeout, int nanos) {
        void = Object
    }
    public ForkJoinWorkerThreadFactory defaultForkJoinWorkerThreadFactory() {
        ForkJoinWorkerThreadFactory = ForkJoinPool
    }
    public void managedBlock(ManagedBlocker blocker) {
        void = ForkJoinPool
    }

```

图 9-4 类 ForkJoinPool 中可调用的方法完整列表

在后面的章节将介绍核心方法的使用。

类 ForkJoinPool 所提供的功能是一个任务池，而执行具体任务却不是 ForkJoinPool，而是 ForkJoinTask 类，类 ForkJoinTask 信息如图 9-5 所示。

类 ForkJoinTask 是抽象类，不能实例化，所以需要该类的 3 个子类 CountedCompleter、RecursiveAction 和 RecursiveTask 来实现具体功能。

从下面的章节开始，将以若干案例来实验本章的主题：分治编程的使用。

```

java.util.concurrent
Class ForkJoinTask<V>
java.lang.Object
    java.util.concurrent.ForkJoinTask<V>

All implemented interfaces:
Serializable, Future<V>

Direct Known Subclasses:
CountedCompleter, RecursiveAction, RecursiveTask

public abstract class ForkJoinTask<V>
    extends Object
    implements Future<V>, Serializable

```

图 9-5 类信息 ForkJoinTask

9.2 使用 RecursiveAction 让任务跑起来

使用类 RecursiveAction 执行的任务是具有无返回值的，仅执行一次任务。

创建项目 test1，创建类 MyRecursiveAction.java 代码如下：

```

package action;

import java.util.concurrent.RecursiveAction;

```

```

public class MyRecursiveAction extends RecursiveAction {
    @Override
    protected void compute() {
        System.out.println("compute run!");
    }
}

```

运行类 Test.java 代码如下:

```

package test;

import java.util.concurrent.ForkJoinPool;
import action.MyRecursiveAction;

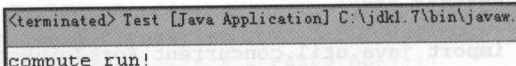
public class Test {

    public static void main(String[] args) throws InterruptedException
    {
        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(new MyRecursiveAction());
        Thread.sleep(5000);
    }
}

```

程序运行结果如图 9-6 所示。

任务成功被运行起来。



```

<terminated> Test [Java Application] C:\jdk1.7\bin\javaw.
compute run!

```

图 9-6 运行结果

9.3 使用 RecursiveAction 分解任务

前面的示例仅是让任务运行起来,并打印一个字符串信息,任务并没有得到 fork 分解,也就是并没有体现分治编程的运行效果。在调用 ForkJoinTask.java 类中的 fork() 方法时需要注意一下效率的问题,因为每一次调用 fork 都会分离任务,增加系统运行负担,所以在 ForkJoinTask.java 类中提供了 public static void invokeAll(ForkJoinTask<?> t1, ForkJoinTask<?> t2) 方法来优化执行效率。本示例也是使用此方法分离任务并执行。

创建项目 test2,创建类 MyRecursiveAction.java 代码如下:

```

package action;

import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction {

    private int beginValue;
    private int endValue;

    public MyRecursiveAction(int beginValue, int endValue) {

```

```

    super();
    this.beginValue = beginValue;
    this.endValue = endValue;
}

@Override
protected void compute() {
    System.out.println(Thread.currentThread().getName() + " -----");
    if (endValue - beginValue > 2) {
        int middelNum = (beginValue + endValue) / 2;
        MyRecursiveAction leftAction = new MyRecursiveAction(beginValue,
            middelNum);
        MyRecursiveAction rightAction = new MyRecursiveAction(
            middelNum + 1, endValue);
        this.invokeAll(leftAction, rightAction);
    } else {
        System.out.println("打印组合为: " + beginValue + "-" + endValue);
    }
}
}
}

```

运行类 Test.java 代码如下:

```

package test;

import java.util.concurrent.ForkJoinPool;
import action.MyRecursiveAction;

public class Test {

    public static void main(String[] args) throws InterruptedException {
        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(new MyRecursiveAction(1, 10));
        Thread.sleep(5000);
    }
}

```

程序运行结果如图 9-7 所示。

任务被成功分解。

```

ForkJoinPool-1-worker-1 -----
ForkJoinPool-1-worker-1 -----
ForkJoinPool-1-worker-1 -----
打印组合为: 1-3
ForkJoinPool-1-worker-1 -----
打印组合为: 4-5
ForkJoinPool-1-worker-2 -----
ForkJoinPool-1-worker-2 -----
ForkJoinPool-1-worker-4 -----
打印组合为: 6-8
打印组合为: 9-10

```

图 9-7 运行结果

9.4 使用 RecursiveTask 取得返回值与 join() 和 get() 方法的区别

使用类 RecursiveTask 执行的任务具有返回值的功能。

创建项目 test3, 创建类 MyRecursiveTask.java 代码如下:

```

package mytask;

import java.util.concurrent.RecursiveTask;

```

```

public class MyRecursiveTask extends RecursiveTask<Integer> {
    @Override
    protected Integer compute() {
        System.out.println("compute time=" + System.currentTimeMillis());
        return 100;
    }
}

```

运行类 Test1.java 代码如下:

```

package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

import mytask.MyRecursiveTask;

public class Test1 {

    public static void main(String[] args) {
        try {
            MyRecursiveTask task1 = new MyRecursiveTask();
            System.out.println(task1.hashCode());
            ForkJoinPool pool = new ForkJoinPool();
            ForkJoinTask task2 = pool.submit(task1);
            System.out.println(task2.hashCode() + " " + task2.get());
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

```

20230270
compute time=1436507685609
20230270 100

```

程序运行结果如图 9-8 所示。

任务成功返回值为 100。

也可以使用 join() 方法来取得结果值, 创建类 Test2.java 代码如下:

```

package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

import mytask.MyRecursiveTask;

public class Test2 {

    public static void main(String[] args) {
        try {

```

图 9-8 运行结果

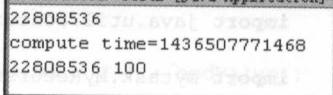
```

        MyRecursiveTask task1 = new MyRecursiveTask();
        System.out.println(task1.hashCode());
        ForkJoinPool pool = new ForkJoinPool();
        ForkJoinTask task2 = pool.submit(task1);
        System.out.println(task2.hashCode() + " " + task2.join());
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

程序运行结果如图 9-9 所示。

方法 join() 与 get() 虽然都能取得计算后的结果值，但它们之间还是在出现异常时有处理上的区别，创建实验用的项目 join_get，创建类 MyRecursiveTask.java 代码如下：



```

22808536
compute time=1436507771468
22808536 100

```

图 9-9 运行结果

```

package mytask;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<Integer> {
    @Override
    protected Integer compute() {
        System.out.println(Thread.currentThread().getName() + " 执行 compute 方法 ()");
        String nullString = null;
        nullString.toString();
        return 100;
    }
}

```

类 Test1.java 代码如下：

```

package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import mytask.MyRecursiveTask;

public class Test1 {

    public static void main(String[] args) {
        try {
            MyRecursiveTask task1 = new MyRecursiveTask();
            ForkJoinPool pool = new ForkJoinPool();
            ForkJoinTask task2 = pool.submit(task1);
            System.out.println(task2.get());
            for (int i = 0; i < Integer.MAX_VALUE; i++) {
                String newString = new String();
                Math.random();
            }
        }
    }
}

```

```

        Math.random();
        Math.random();
        Math.random();
        Math.random();
        Math.random();
        Math.random();
        Math.random();
    }
} catch (InterruptedException e) {
    e.printStackTrace();
    System.out.println(" 进入了 mainA");
} catch (ExecutionException e) {
    e.printStackTrace();
    System.out.println(" 进入了 mainB");
}
System.out.println("main end");
}
}

```

程序运行结果如图 9-10 所示。

```

ForkJoinPool-1-worker-1 执行compute方法()
java.util.concurrent.ExecutionException: java.lang.NullPointerException
    at java.util.concurrent.ForkJoinTask.get(ForkJoinTask.java:942)
    at Test1.main(Test1.java:16)
Caused by: java.lang.NullPointerException
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:57)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:526)
    at java.util.concurrent.ForkJoinTask.getThrowableException(ForkJoinTask.java:536)
    at java.util.concurrent.ForkJoinTask.get(ForkJoinTask.java:941)
    ... 1 more
Caused by: java.lang.NullPointerException
    at mytask.MyRecursiveTask.compute(MyRecursiveTask.java:10)
    at mytask.MyRecursiveTask.compute(MyRecursiveTask.java:1)
    at java.util.concurrent.RecursiveTask.exec(RecursiveTask.java:93)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:334)
    at java.util.concurrent.ForkJoinWorkerThread.execTask(ForkJoinWorkerThread.java:604)
    at java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:784)
    at java.util.concurrent.ForkJoinPool.work(ForkJoinPool.java:646)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:398)
进入了mainB
main end

```

图 9-10 使用方法 get() 在出现异常时进入 catch

使用 get() 方法执行任务时，当子任务出现异常时可以在 main 主线程中进行捕获。

类 Test2.java 代码如下：

```

package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

import mytask.MyRecursiveTask;

public class Test2 {

    public static void main(String[] args) {
        MyRecursiveTask task1 = new MyRecursiveTask();
        ForkJoinPool pool = new ForkJoinPool();
    }
}

```



```

ForkJoinTask task2 = pool.submit(task1);
System.out.println(task2.join());
for (int i = 0; i < Integer.MAX_VALUE; i++) {
    String newString = new String();
    Math.random();
    Math.random();
    Math.random();
    Math.random();
    Math.random();
    Math.random();
    Math.random();
    Math.random();
}
System.out.println("main end");
}
}

```

程序运行结果如图 9-11 所示。

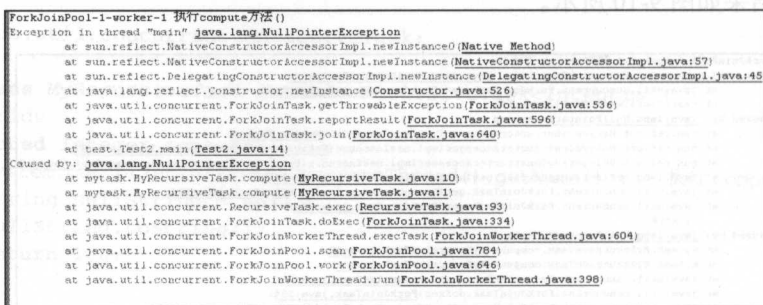


图 9-11 方法 join() 遇到异常直接抛出

9.5 使用 RecursiveTask 执行多个任务并打印返回值

创建项目 test4，创建类 MyRecursiveTaskA.java 代码如下：

```

package task;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTaskA extends RecursiveTask<Integer> {

    @Override
    protected Integer compute() {
        try {
            System.out.println(Thread.currentThread().getName() + " begin A "
                + System.currentTimeMillis());
            Thread.sleep(3000);
            System.out.println(Thread.currentThread().getName() + " end A "

```

```

        + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return 100;
}
}

```

创建类 MyRecursiveTaskB.java 代码如下:

```

package task;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTaskB extends RecursiveTask<Integer> {

    @Override
    protected Integer compute() {
        try {
            System.out.println(Thread.currentThread().getName() + " begin B "
                + System.currentTimeMillis());
            Thread.sleep(5000);
            System.out.println(Thread.currentThread().getName() + " end B "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return 100;
    }
}

```

运行类 Test.java 代码如下:

```

package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

import task.MyRecursiveTaskA;
import task.MyRecursiveTaskB;

public class Test {

    public static void main(String[] args) throws InterruptedException {
        ForkJoinPool pool = new ForkJoinPool();
        ForkJoinTask<Integer> runTaskA = pool.submit(new MyRecursiveTaskA());
        ForkJoinTask<Integer> runTaskB = pool.submit(new MyRecursiveTaskB());
        System.out.println("准备打印: " + System.currentTimeMillis());
        System.out
            .println(runTaskA.join() + " A: " + System.currentTimeMillis());
        System.out
            .println(runTaskB.join() + " B: " + System.currentTimeMillis());
    }
}

```

```

        Thread.sleep(5000);
    }
}

```

程序运行结果如图 9-12 所示。

每个任务成功返回值为 100，并且任务之间运行的方式是异步的，但 `join()` 方法却是同步的。

```

准备打印: 1432007913843
ForkJoinPool-1-worker-2 begin B 1432007913843
ForkJoinPool-1-worker-1 begin A 1432007913843
ForkJoinPool-1-worker-1 end A 1432007916843
100 A 1432007916843
ForkJoinPool-1-worker-2 end B 1432007918843
100 B 1432007918843

```

图 9-12 运行结果

9.6 使用 RecursiveTask 实现字符串累加

创建项目 test5，创建类 `MyRecursiveTask.java` 代码如下：

```

package task;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<String> {

    private int beginValue;
    private int endValue;

    public MyRecursiveTask(int beginValue, int endValue) {
        this.beginValue = beginValue;
        this.endValue = endValue;
    }

    @Override
    protected String compute() {
        System.out.println(Thread.currentThread().getName() + " -----");
        if (endValue - beginValue > 2) {
            int midValue = (endValue + beginValue) / 2;

            MyRecursiveTask leftTask = new MyRecursiveTask(beginValue,
                midValue);
            MyRecursiveTask rightTask = new MyRecursiveTask(midValue + 1,
                endValue);

            this.invokeAll(leftTask, rightTask);

            return leftTask.join() + rightTask.join();
        } else {
            String returnString = "";
            for (int i = beginValue; i <= endValue; i++) {
                returnString = returnString + (i);
            }
            System.out.println("else 返回: " + returnString + " " + beginValue
                + " " + endValue);
            return returnString;
        }
    }
}

```

本示例完全可以比喻成蜂王命令小蜜蜂去采蜜，代码 `return leftTask.join() + rightTask.join()`；相当于将每个小蜜蜂采的蜜进行汇总，而代码 `return returnString`；是取得每个小蜜蜂采的蜜。

运行类 `Test.java` 代码如下：

```
package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

import task.MyRecursiveTask;

public class Test {

    public static void main(String[] args) throws InterruptedException {
        ForkJoinPool pool = new ForkJoinPool();
        MyRecursiveTask task = new MyRecursiveTask(1, 20);
        ForkJoinTask<String> runTaskA = pool.submit(task);
        System.out.println(runTaskA.join());
        Thread.sleep(5000);
    }
}
```

程序运行结果如图 9-13 所示。

成功返回累加的字符串 1234567891011121314151617181920。

9.7 使用 Fork-Join 实现求和：实验 1

创建测试用的项目 `forkjoin_2`，类 `MyRecursiveTask.java` 代码如下：

```
package mytask;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<Integer> {

    private int beginPosition;
    private int endPosition;

    public MyRecursiveTask(int beginPosition, int endPosition) {
        super();
        this.beginPosition = beginPosition;
        this.endPosition = endPosition;
        System.out.println("# " + (beginPosition + " " + endPosition));
    }

    protected Integer compute() {
        System.out.println(Thread.currentThread().getName())
```

```
ForkJoinPool-1-worker-1 -----
ForkJoinPool-1-worker-1 -----
ForkJoinPool-1-worker-1 -----
ForkJoinPool-1-worker-2 -----
ForkJoinPool-1-worker-3 -----
ForkJoinPool-1-worker-1 -----
ForkJoinPool-1-worker-3 -----
ForkJoinPool-1-worker-2 -----
ForkJoinPool-1-worker-4 -----
ForkJoinPool-1-worker-2 -----
else返回: 678      6 8
else返回: 123      1 3
else返回: 111213   11 13
ForkJoinPool-1-worker-4 -----
ForkJoinPool-1-worker-2 -----
ForkJoinPool-1-worker-3 -----
ForkJoinPool-1-worker-1 -----
else返回: 910      9 10
else返回: 1415     14 15
else返回: 161718   16 18
ForkJoinPool-1-worker-3 -----
else返回: 45        4 5
else返回: 1920     19 20
1234567891011121314151617181920
```

图 9-13 运行结果

```

+ "-----");
Integer sumValue = 0;
System.out.println("compute=" + beginPosition + " " + endPosition);
if ((endPosition - beginPosition) != 0) {
    System.out.println("!=0");
    int middleNum = (endPosition + beginPosition) / 2;
    System.out.println("left 传入的值："
        + (beginPosition + " " + middleNum));
    MyRecursiveTask leftTask = new MyRecursiveTask(beginPosition,
        middleNum);
    System.out.println("right 传入的值："
        + ((middleNum + 1) + " " + endPosition));
    MyRecursiveTask rightTask = new MyRecursiveTask(middleNum + 1,
        endPosition);

    this.invokeAll(leftTask, rightTask);

    Integer leftValue = leftTask.join();
    Integer rightValue = rightTask.join();

    return leftValue + rightValue;
} else {
    return endPosition;
}
}
}

```

本实验的核心条件是 `if((endPosition-beginPosition)!=0)` 代码，也就是想实现以 `1 + 2 + 3 + 4 + 5 + ……` 的方式累加到 10。

类 `Run.java` 代码如下：

```

package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;

import mytask.MyRecursiveTask;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        try {
            MyRecursiveTask task = new MyRecursiveTask(1, 10);
            ForkJoinPool pool = new ForkJoinPool();
            pool.submit(task);
            System.out.println("结果值为：" + task.get());
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 9-14 所示。

9.8 使用 Fork-Join 实现求和：实验 2

创建测试用的项目 forkjoin_3，类 MyRecursiveTask.java 代码如下：

```
package mytask;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<Integer> {

    private int beginPosition;
    private int endPosition;

    public MyRecursiveTask(int beginPosition, int endPosition) {
        super();
        this.beginPosition = beginPosition;
        this.endPosition = endPosition;
        System.out.println("# " + (beginPosition + " " + endPosition));
    }

    protected Integer compute() {
        Integer sumValue = 0;
        System.out.println("compute=" + beginPosition + " " + endPosition);
        if ((endPosition - beginPosition) > 2) {
            System.out.println("!=0");
            int middleNum = (endPosition + beginPosition) / 2;
            System.out.println("left 传入的值：" + (beginPosition + " " + middleNum));
            MyRecursiveTask leftTask = new MyRecursiveTask(beginPosition, middleNum);
            System.out.println("right 传入的值：" + ((middleNum + 1) + " " + endPosition));
            MyRecursiveTask rightTask = new MyRecursiveTask(middleNum + 1, endPosition);

            this.invokeAll(leftTask, rightTask);

            Integer leftValue = leftTask.join();
            Integer rightValue = rightTask.join();
            return leftValue + rightValue;
        } else {
            int count = 0;
            for (int i = beginPosition; i <= endPosition; i++) {
                count = count + i;
            }
        }
    }
}
```

```
right 传入的值:10 10
ForkJoinPool-1-worker-3-----
# 10 10
left 传入的值:6 6
right 传入的值:3 3
# 6 6
ForkJoinPool-1-worker-4-----
compute=8 8
compute=9 9
right 传入的值:7 7
# 7 7
# 3 3
ForkJoinPool-1-worker-2-----
ForkJoinPool-1-worker-3-----
compute=6 6
ForkJoinPool-1-worker-1-----
ForkJoinPool-1-worker-2-----
compute=10 10
compute=7 7
compute=1 2
!=0
left 传入的值:1 1
ForkJoinPool-1-worker-3-----
compute=3 3
# 1 1
right 传入的值:2 2
# 2 2
ForkJoinPool-1-worker-1-----
compute=1 1
ForkJoinPool-1-worker-1-----
compute=2 2
结果值为: 55
```

图 9-14 分解再合并成功取出求和值的部分打印结果


```

    }
    return count;
}
}
}

```

与 9.7 节中的代码不同，本实验的核心条件是 `if ((endPosition-beginPosition) > 2)` 代码，也就是可以使用 `else{}` 中的 `for` 循环以数字范围的方式进行累加求和。

类 `Run.java` 代码如下：

```

package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;

import mytask.MyRecursiveTask;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        try {
            MyRecursiveTask task = new MyRecursiveTask(1, 10);
            ForkJoinPool pool = new ForkJoinPool();
            System.out.println(" 结果值为: " +
                pool.submit(task).get());
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 9-15 所示。

```

# 1 10
compute=1 10
!=0
left 传入的值:1 5
# 1 5
right 传入的值:6 10
# 6 10
compute=1 5
!=0
left 传入的值:1 3
# 1 3
right 传入的值:4 5
compute=6 10
!=0
# 4 5
left 传入的值:6 8
# 6 8
right 传入的值:9 10
compute=1 3
compute=4 5
# 9 10
compute=6 8
compute=9 10
结果值为: 55

```

9.9 类 `ForkJoinPool` 核心方法的实验

完整的 `ForkJoinPool` 对象可调用方法列表如图 9-16 所示。

在后面的章节将介绍核心方法的使用。

图 9-15 分解再合并成功取出求和值

9.9.1 方法 `public void execute(ForkJoinTask<?> task)` 的使用

在 `ForkJoinPool.java` 类中的 `execute()` 方法是以异步的方式执行任务。

创建名称为 `method1` 的 Java 项目，创建类 `MyRecursiveAction.java` 代码如下：

```

package myaction;

import java.util.concurrent.RecursiveAction;

```

```

public class MyRecursiveAction extends RecursiveAction {
    @Override
    protected void compute() {
        System.out.println("ThreadName=" + Thread.currentThread().getName());
    }
}

```

```

awaitTermination(long timeout, TimeUnit unit) : boolean - ForkJoinPool
equals(Object obj) : boolean - Object
execute(ForkJoinTask<?> task) : void - ForkJoinPool
execute(Runnable task) : void - ForkJoinPool
getActiveThreadCount() : int - ForkJoinPool
getAsyncMode() : boolean - ForkJoinPool
getClass() : Class<?> - Object
getFactory() : ForkJoinWorkerThreadFactory - ForkJoinPool
getParallelism() : int - ForkJoinPool
getPoolSize() : int - ForkJoinPool
getQueuedSubmissionCount() : int - ForkJoinPool
getQueuedTaskCount() : long - ForkJoinPool
getRunningThreadCount() : int - ForkJoinPool
getStealCount() : long - ForkJoinPool
getUncaughtExceptionHandler() : UncaughtExceptionHandler - ForkJoinPool
hashCode() : int - Object
hasQueuedSubmissions() : boolean - ForkJoinPool
invoke(ForkJoinTask<T> task) : T - ForkJoinPool
invokeAll(Collection<? extends Callable<T>> tasks) : List<Future<T>> - ForkJoinPool
invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) : List<Future<T>> - AbstractExecutorService
invokeAny(Collection<? extends Callable<T>> tasks) : T - AbstractExecutorService
invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) : T - AbstractExecutorService
isQuiescent() : boolean - ForkJoinPool
isShutdown() : boolean - ForkJoinPool
isTerminated() : boolean - ForkJoinPool
isTerminating() : boolean - ForkJoinPool
notify() : void - Object
notifyAll() : void - Object
shutdown() : void - ForkJoinPool
shutdownNow() : List<Runnable> - ForkJoinPool
submit(Callable<T> task) : ForkJoinTask<T> - ForkJoinPool
submit(ForkJoinTask<T> task) : ForkJoinTask<T> - ForkJoinPool
submit(Runnable task) : ForkJoinTask<?> - ForkJoinPool
submit(Runnable task, T result) : ForkJoinTask<T> - ForkJoinPool
toString() : String - ForkJoinPool
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
defaultForkJoinWorkerThreadFactory : ForkJoinPool.ForkJoinWorkerThreadFactory - ForkJoinPool
managedBlock(ManagedBlocker blocker) : void - ForkJoinPool

```

图 9-16 类 ForkJoinPool 中可调用的方法完整列表

运行类 Run.java 代码如下:

```

package test;

import java.util.concurrent.ForkJoinPool;
import myaction.MyRecursiveAction;

public class Run {

```

```

    public static void main(String[] args) {
        try {

```

```

            ForkJoinPool pool = new ForkJoinPool();
            pool.execute(new MyRecursiveAction());

```

```

        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

```

<terminated> Run [Java Application] C:\jdk1.7\bin\jav
ThreadName=ForkJoinPool-1-worker-1

```

程序运行结果如图 9-17 所示。

图 9-17 运行结果

任务成功被运行。

9.9.2 方法 `public void execute(Runnable task)` 的使用

创建名称为 method2 的 Java 项目，运行类 Run.java 代码如下：

```

package test;

import java.util.concurrent.ForkJoinPool;

public class Run {

    public static void main(String[] args) {
        try {
            ForkJoinPool pool = new ForkJoinPool();
            pool.execute(new Runnable() {
                @Override
                public void run() {
                    System.out.println("ThreadName="
                        + Thread.currentThread().getName());
                }
            });
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

```

<terminated> Run [Java Application] C:\jdk1.7\bin\jav
ThreadName=ForkJoinPool-1-worker-1

```

程序运行结果如图 9-18 所示。

图 9-18 运行结果

任务成功被运行。

9.9.3 方法 `public void execute(ForkJoinTask<?> task)` 如何处理返回值

创建名称为 method3 的 Java 项目，创建类 MyRecursiveTask.java 代码如下：

```

package mytask;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<String> {
    // ...
}

```

```

@Override
protected String compute() {
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "我是返回值";
}
}

```

运行类 Run.java 代码如下:

```

package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import mytask.MyRecursiveTask;

public class Run {

    public static void main(String[] args) {
        try {
            MyRecursiveTask task = new MyRecursiveTask();
            ForkJoinPool pool = new ForkJoinPool();
            pool.execute(task);
            // execute 方法无返回值
            // 想取得返回值得通过
            // RecursiveTask 对象
            System.out.println(System.currentTimeMillis());
            System.out.println(task.get());
            System.out.println(System.currentTimeMillis());
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

```

<terminated> Run (1) [Java Application] C:\jdk1.7
ThreadName=ForkJoinPool-1-worker-1

```

程序运行结果如图 9-19 所示。

图 9-19 成功取得返回值

虽然 `public void execute(ForkJoinTask<?> task)` 方法无返回值,但还是可以通过 `RecursiveTask` 对象处理返回值。

9.9.4 方法 `public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task)` 的使用

方法 `execute()` 无返回值, `submit()` 有返回值。

创建名称为 `method4` 的 Java 项目,创建类 `MyRecursiveTask.java` 代码如下:

```

package mytask;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<String> {

    @Override
    protected String compute() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "我是返回值";
    }
}

```

运行类 Run.java 代码如下：

```

package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import mytask.MyRecursiveTask;

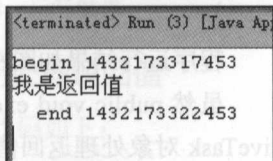
public class Run {

    public static void main(String[] args) {
        try {
            MyRecursiveTask task = new MyRecursiveTask();
            ForkJoinPool pool = new ForkJoinPool();
            ForkJoinTask<String> returnTask = pool.submit(task);
            System.out.println("begin " + System.currentTimeMillis());
            System.out.println(returnTask.get());
            System.out.println(" end " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 9-20 所示。

任务成功运行。



```

<terminated> Run (3) [Java Ap
begin 1432173317453
我是返回值
end 1432173322453

```

图 9-20 运行结果

9.9.5 方法 public ForkJoinTask<?> submit(Runnable task) 的使用

创建名称为 method5 的 Java 项目，运行类 Run.java 代码如下：

```

package test;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

public class Run {

    public static void main(String[] args) {
        try {
            ForkJoinPool pool = new ForkJoinPool();
            System.out.println("begin " + System.currentTimeMillis());
            ForkJoinTask task = pool.submit(new Runnable() {
                @Override
                public void run() {
                    try {
                        Thread.sleep(5000);
                        System.out.println("ThreadName="
                            + Thread.currentThread().getName());
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
            System.out.println(task.get());
            System.out.println("end " + System.currentTimeMillis());
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

```

<terminated> Run (4) [Java Application] C:\jdk1
begin 1432173405921
ThreadName=ForkJoinPool-1-worker-1
null
end 1432173410921

```

图 9-21 运行结果

程序运行结果如图 9-21 所示。

任务成功被运行，传入 Runnable 接口虽然没有返回值，但调用 get() 方法呈阻塞状态。

9.9.6 方法 public <T> ForkJoinTask<T> submit(Callable<T> task) 的使用

创建名称为 method6 的 Java 项目，创建类 MyCallable.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        try {

```



```

        Thread.sleep(3000);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "我是返回值 callableVersion";
}
}
}

```

运行类 Run.java 代码如下：

```

package test;

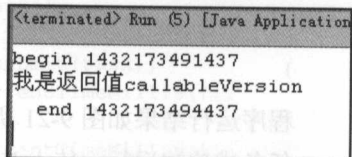
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import mycallable.MyCallable;

public class Run {
    public static void main(String[] args) {
        try {
            MyCallable callable = new MyCallable();
            ForkJoinPool pool = new ForkJoinPool();
            System.out.println("begin " + System.currentTimeMillis());
            ForkJoinTask<String> returnTask = pool.submit(callable);
            System.out.println(returnTask.get());
            System.out.println(" end " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 9-22 所示。

任务成功运行。



```

<terminated> Run (5) [Java Application]
begin 1432173491437
我是返回值callableVersion
end 1432173494437

```

图 9-22 运行结果

9.9.7 方法 `public <T> ForkJoinTask<T> submit(Runnable task, T result)` 的使用

创建名称为 method7 的 Java 项目，创建类 Userinfo.java 代码如下：

```

package entity;

public class Userinfo {
    private String username;

    public Userinfo() {
        super();
    }
}

```

```

    }

    public Userinfo(String username) {
        super();
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}

```

创建类 MyRunnable.java 代码如下:

```

package myrunnable;

import entity.Userinfo;

public class MyRunnable implements Runnable {

    private Userinfo userinfo;

    public MyRunnable(Userinfo userinfo) {
        super();
        this.userinfo = userinfo;
    }

    public void run() {
        try {
            userinfo.setUsername(" 设置的值 ");
            System.out.println(" 已经设置完结! ");
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

运行类 Run1.java 代码如下:

```

package test.run;

import java.util.concurrent.ForkJoinPool;

import myrunnable.MyRunnable;
import entity.Userinfo;

```

```

public class Run1 {
    public static void main(String[] args) throws InterruptedException {
        Userinfo userinfo = new Userinfo();
        MyRunnable runnable = new MyRunnable(userinfo);

        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(runnable, userinfo);
        // 取不到值
        System.out.println("userinfo username=" + userinfo.getUsername());
        Thread.sleep(2000);
    }
}

```

程序运行结果如图 9-23 所示。

运行结果是未取到值，因为是异步运行的，所以要加一个延时功能。

创建类 Run2.java 代码如下：

```

package test.run;

import java.util.concurrent.ForkJoinPool;

import myrunnable.MyRunnable;
import entity.Userinfo;

public class Run2 {

    public static void main(String[] args) throws InterruptedException {
        Userinfo userinfo = new Userinfo();
        MyRunnable runnable = new MyRunnable(userinfo);

        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(runnable, userinfo);
        Thread.sleep(2000);
        // 上面使用 sleep 不标准，因为执行时间不固定
        // 如果打印的早则有可能出现打印 null 值的情况
        System.out.println("userinfo username=" + userinfo.getUsername());
        Thread.sleep(2000);
    }
}

```

程序运行结果如图 9-24 所示。

成功取到值，但程序还是有一些问题，也就是 sleep() 的时间并不能真正的确定，因为不知道 Task 任务执行完需要多少时间，更改代码如 Run3.java 所示：

```

package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;

```

图 9-23 运行结果

图 9-24 运行结果

```

import myrunnable.MyRunnable;
import entity.UserInfo;

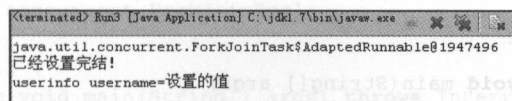
public class Run3 {

    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        UserInfo userinfo = new UserInfo();
        MyRunnable runnable = new MyRunnable(userinfo);

        ForkJoinPool pool = new ForkJoinPool();
        Future<UserInfo> future = pool.submit(runnable, userinfo);
        System.out.println(future);
        // 建议使用此种方式 future.get()
        // 因为 get() 方法呈阻塞效果
        System.out.println("userinfo username=" + future.get().getUsername());
    }
}

```

程序运行结果如图 9-25 所示。



```

terminated> Run3 [Java Application] C:\jdk1.7\bin\java.exe
java.util.concurrent.ForkJoinTask$AdaptedRunnable@1947496
已经设置完结!
userinfo username=设置的值

```

图 9-25 运行结果

9.9.8 方法 `public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)` 的使用

方法 `public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)` 具有阻塞特性。

创建名称为 method15 的 Java 项目，创建类 MyCallable.java 代码如下：

```

package mycallable;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    private long sleepValue;

    public MyCallable(long sleepValue) {
        super();
        this.sleepValue = sleepValue;
    }

    @Override
    public String call() throws Exception {
        try {
            System.out.println(Thread.currentThread().getName() + " sleep"
                + sleepValue + " nowTime: " + System.currentTimeMillis());

```

```

        Thread.sleep(sleepValue);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "我是返回值";
}
}

```

运行类 Test.java 代码如下：

```

package test;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;

import mycallable.MyCallable;

public class Test {

    public static void main(String[] args) {
        try {
            List list = new ArrayList();
            list.add(new MyCallable(5000));
            list.add(new MyCallable(4000));
            list.add(new MyCallable(3000));
            list.add(new MyCallable(2000));
            list.add(new MyCallable(1000));

            ForkJoinPool pool = new ForkJoinPool();
            List<Future<String>> listFuture = pool.invokeAll(list);
            for (int i = 0; i < listFuture.size(); i++) {
                System.out.println(listFuture.get(i).get() + " nowTime: "
                    + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 9-26 所示。

9.9.9 方法 public void shutdown() 的使用

创建名称为 method8 的 Java 项目，创建类

MyRunnable1.java 代码如下：

```

ForkJoinPool-1-worker-3 sleep1000 nowTime: 1435112084093
ForkJoinPool-1-worker-1 sleep5000 nowTime: 1435112084093
ForkJoinPool-1-worker-4 sleep3000 nowTime: 1435112084093
ForkJoinPool-1-worker-2 sleep2000 nowTime: 1435112084093
ForkJoinPool-1-worker-3 sleep4000 nowTime: 1435112085093
我是返回值 nowTime: 1435112089093
我是返回值 nowTime: 1435112089093
我是返回值 nowTime: 1435112089093
我是返回值 nowTime: 1435112089093
我是返回值 nowTime: 1435112089093

```

图 9-26 运行结果

```

package myrunnable;

public class MyRunnable1 implements Runnable {
    public void run() {
        try {
            System.out.println("begin " + Thread.currentThread().getName()
                + " " + System.currentTimeMillis());
            Thread.sleep(4000);
            System.out.println(" end " + Thread.currentThread().getName()
                + " " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

运行类 Test1.java 代码如下:

```

package test;

import java.util.concurrent.ForkJoinPool;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        ForkJoinPool pool = new ForkJoinPool();
        System.out.println("main end!");
    }
}

```

程序运行结果如图 9-27 所示。

程序运行打印出“main end!”后进程立即销毁。

运行类 Test2.java 代码如下:

```

package test;

import java.util.concurrent.ForkJoinPool;
import myrunnable.MyRunnable1;

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(myRunnable);
        Thread.sleep(1000);
        pool.shutdown();
        System.out.println("main end!");
        Thread.sleep(Integer.MAX_VALUE);
    }
}

```

程序运行结果如图 9-28 所示。

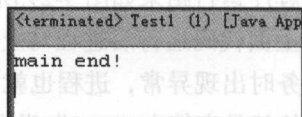


图 9-27 运行结果

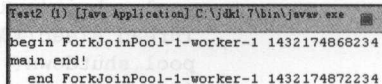


图 9-28 运行结果

从运行结果来看，调用 shutdown() 方法后任务依然运行，直到结束。
运行类 Test3.java 代码如下：

```
package test;

import java.util.concurrent.ForkJoinPool;

import myrunnable.MyRunnable1;

public class Test3 {
    public static void main(String[] args) {
        try {
            MyRunnable1 myRunnable = new MyRunnable1();
            ForkJoinPool pool = new ForkJoinPool();
            pool.submit(myRunnable);
            Thread.sleep(1000);
            pool.shutdown();
            pool.submit(myRunnable);
            System.out.println("main end!");
            Thread.sleep(Integer.MAX_VALUE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 9-29 所示。

上面代码运行后进程马上被销毁，说明对 ForkJoinPool 对象调用 shutdown() 方法后再执行任务时出现异常，进程也就马上销毁了，而正在运行的线程任务也被销毁掉了，运行的效果当然就是字符串“end”没有输出，仅仅打印了“begin”。

为了防止在关闭 pool 后再运行任务，可以加入一个判断来解决进程意外销毁的问题。
继续实验，创建 Test4.java 类，代码如下：

```
package test;

import java.util.concurrent.ForkJoinPool;

import myrunnable.MyRunnable1;

public class Test4 {
    public static void main(String[] args) {
        try {
            MyRunnable1 myRunnable = new MyRunnable1();
            ForkJoinPool pool = new ForkJoinPool();
            pool.submit(myRunnable);
            Thread.sleep(1000);
            pool.shutdown();
            if (pool.isShutdown() == false) {
                pool.submit(myRunnable);
            }
        }
    }
}
```

```

    System.out.println("main end!");
    Thread.sleep(Integer.MAX_VALUE);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

```

程序运行结果如图 9-30 所示。

```

begin ForkJoinPool-1-worker-1 1432174919328
Exception in thread "main" java.util.concurrent.RejectedExecutionException
at java.util.concurrent.ForkJoinPool.submit(ForkJoinPool.java:1533)
at java.util.concurrent.ForkJoinPool.submit(ForkJoinPool.java:1628)
at test.Test3.main(Test3.java:17)

```

图 9-29 运行结果

```

Test4 [Java Application] C:\jdk1.7\bin\javaw.exe
begin ForkJoinPool-1-worker-1 1434682476359
main end!
end ForkJoinPool-1-worker-1 1434682480359

```

图 9-30 运行结果

进程并没有被销毁，而任务也成功打印了字符串“end”。

通过本示例还可以发现一个小知识点，就是 shutdown() 方法不具有中断的效果，也就是 shutdown() 方法遇到 MyRunnable1.java 类中的 sleep() 方法并没有发生中断异常。

9.9.10 方法 public List<Runnable> shutdownNow() 的使用

创建名称为 method9 的 Java 项目，创建类 MyRunnable1.java 代码如下：

```
package myrunnable;
```

```

public class MyRunnable1 implements Runnable {
    public void run() {
        try {
            for (int i = 0; i < Integer.MAX_VALUE; i++) {
                String newString = new String();
                Math.random();
                Math.random();
                Math.random();
                Math.random();
                Math.random();
                Math.random();

                if (Thread.currentThread().isInterrupted() == true) {
                    System.out.println("任务没有完成，就中断了!");
                    throw new InterruptedException();
                }
            }
            System.out.println("任务成功完成!");
        } catch (InterruptedException e) {
            System.out.println("进入 catch 中断了任务");
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 9-33 所示。

图 9-31 运行结果

图 9-32 运行结果

创建类 MyRunnable2.java 代码如下：

```
package myrunnable;

public class MyRunnable2 implements Runnable {
    public void run() {
        for (int i = 0; i < Integer.MAX_VALUE / 100; i++) {
            new String();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
        }
        System.out.println(" 任务成功完成！ ");
    }
}
```

运行类 Test0.java 代码如下：

```
package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test0 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(myRunnable);
        Thread.sleep(2000);
        pool.shutdownNow();
        System.out.println("main end");
        Thread.sleep(Integer.MAX_VALUE);
    }
}
```

程序运行结果如图 9-31 所示。

```
main end
java.lang.InterruptedException
    at myrunnable.MyRunnable1.run(MyRunnable1.java:17)
    at java.util.concurrent.ForkJoinTask$AdaptedRunnable.exec(ForkJoinTask.java:1265)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:334)
    at java.util.concurrent.ForkJoinWorkerThread.execTask(ForkJoinWorkerThread.java:604)
    at java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:784)
    at java.util.concurrent.ForkJoinPool.work(ForkJoinPool.java:646)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:398)
任务没有完成，就中断了！
进入catch中断了任务
```

图 9-31 运行结果

运行类 Test1.java 代码如下:

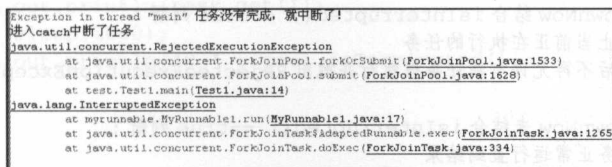
```
package test;

import java.util.concurrent.ForkJoinPool;

import myrunnable.MyRunnable1;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(myRunnable);
        Thread.sleep(2000);
        pool.shutdownNow();
        pool.submit(myRunnable);
    }
}
```

程序运行结果如图 9-32 所示。



```
Exception in thread "main": 任务没有完成, 就中断了!
进入catch中断了任务
java.util.concurrent.RejectedExecutionException
    at java.util.concurrent.ForkJoinPool.ForkJoinTask.awaitJoin(ForkJoinPool.java:1533)
    at java.util.concurrent.ForkJoinPool.submit(ForkJoinPool.java:1628)
    at test.Test1.main(Test1.java:14)
java.lang.InterruptedException
    at myrunnable.MyRunnable1.run(MyRunnable1.java:17)
    at java.util.concurrent.ForkJoinTask$AdaptedRunnable.exec(ForkJoinTask.java:1265)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:334)
```

图 9-32 运行结果

运行类 Test2.java 代码如下:

```
package test;

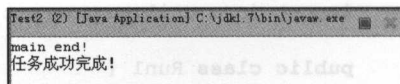
import java.util.concurrent.ForkJoinPool;

import myrunnable.MyRunnable2;

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable2 myRunnable = new MyRunnable2();
        ForkJoinPool pool = new ForkJoinPool();
        pool.execute(myRunnable);
        Thread.sleep(1000);
        pool.shutdownNow();
        System.out.println("main end!");
        Thread.sleep(Integer.MAX_VALUE);
    }
}
```

程序运行结果如图 9-33 所示。

运行类 Test3.java 代码如下:



```
Test2 (2) [Java Application] C:\jdk1.7\bin\java.exe
main end!
任务成功完成!
```

图 9-33 运行结果

```

package test;

import java.util.concurrent.ForkJoinPool;

import myrunnable.MyRunnable2;

public class Test3 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable2 myRunnable = new MyRunnable2();
        ForkJoinPool pool = new ForkJoinPool();
        pool.execute(myRunnable);
        Thread.sleep(1000);
        pool.shutdownNow(); // 返回一个空的 List
        pool.execute(myRunnable);
        System.out.println("main end!");
        Thread.sleep(Integer.MAX_VALUE);

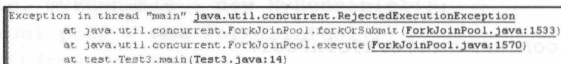
        // shutdown:
        // 每个任务正常运行直到结束,
        // 池关闭后不再允许有新任务被执行并抛出 RejectedExecutionException 异常。

        // shutdownNow 结合 isInterrupted() ==true 判断:
        // 立即停止当前正在执行的任务
        // 池关闭后不再允许有新任务被执行并抛出 RejectedExecutionException 异常

        // shutdownNow 未结合 isInterrupted() ==true 判断:
        // 每个任务正常运行直到结束
        // 池关闭后不再允许有新任务被执行并抛出 RejectedExecutionException 异常
    }
}

```

程序运行结果如图 9-34 所示。



```

Exception in thread "main" java.util.concurrent.RejectedExecutionException
    at java.util.concurrent.ForkJoinPool.ForkJoinTask.awaitJoin(ForkJoinPool.java:1533)
    at java.util.concurrent.ForkJoinPool.execute(ForkJoinPool.java:1570)
    at test.Test3.main(Test3.java:14)

```

图 9-34 运行结果

9.9.11 方法 isTerminating() 和 isTerminated() 的使用

创建名称为 method10 的 Java 项目，创建类 Run1.java 代码如下：

```

package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

public class Run1 {
    public static void main(String[] args) throws InterruptedException,
        ExecutionException {

```

```
ForkJoinPool pool = new ForkJoinPool();
ForkJoinTask task = pool.submit(runnable);
Thread.sleep(500);
System.out.println("A=" + pool.isTerminating());
pool.shutdown();
System.out.println("B=" + pool.isTerminating());
System.out.println(task.get());
Thread.sleep(1000);
System.out.println("C=" + pool.isTerminated());
```

```
public class Run2 {  
    public static void main(String[] args) throws InterruptedException,  
        ExecutionException {  
        Runnable runnable = new Runnable() {  
            @Override  
            public void run() {  
                for (int i = 0; i < Integer.MAX_VALUE / 100; i++) {  
                    String newString = new String();  
                    Math.random();  
                    Math.random();  
                    Math.random();  
                    Math.random();  
                    Math.random();  
                }  
            }  
        };  
        ExecutorService executorService = Executors.newFixedThreadPool(10);  
        List<Future> futures = executorService.invokeAll(Arrays.asList(runnable));  
        for (Future future : futures) {  
            try {  
                future.get();  
            } catch (InterruptedException | ExecutionException e) {}  
        }  
    }  
}
```



```

        Math.random();
    }
}

};

ForkJoinPool pool = new ForkJoinPool();
ForkJoinTask task = pool.submit(runnable);
Thread.sleep(500);
System.out.println("A=" + pool.isTerminating());
pool.shutdownNow();
System.out.println("B=" + pool.isTerminating());
System.out.println(task.get());
Thread.sleep(1000);
System.out.println("C=" + pool.isTerminated());
}
}

```

程序运行结果如图 9-36 所示。

```

<terminated> Run2 (1) [Java Application] C:\jdk1.7\bin\javaw
A=false
Exception in thread "main" B=true
java.util.concurrent.CancellationException
    at java.util.concurrent.ForkJoinTask.get(ForkJoinTask.java:940)
    at test.run.Run2.main(Run2.java:31)

```

图 9-36 运行结果

注意：在控制台输出了一个 `java.util.concurrent.CancellationException` 异常信息，说明先调用 `shutdown()`，再调用 `get()` 方法不出现异常，而先调用 `shutdownNow()`，再调用 `get()` 方法出现异常 `CancellationException`，说明方法 `shutdown()` 与 `shutdownNow()` 在对 `get()` 方法的处理行为上是不一样的。

创建类 `Run3.java` 代码如下：

```

package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

public class Run3 {
    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < Integer.MAX_VALUE / 100; i++) {
                    String newString = new String();
                    Math.random();
                    Math.random();
                    Math.random();
                    Math.random();
                }
            }
        };
    }
}

```

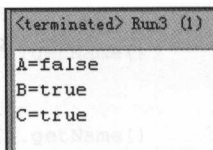
```

        Math.random();
        Math.random();
    }
}

ForkJoinPool pool = new ForkJoinPool();
ForkJoinTask task = pool.submit(runnable);
Thread.sleep(500);
System.out.println("A=" + pool.isTerminating());
pool.shutdownNow();
System.out.println("B=" + pool.isTerminating());
Thread.sleep(30000);
System.out.println("C=" + pool.isTerminated());
}
}

```

程序运行结果如图 9-37 所示。



```

<terminated> Run3 (1)
A=false
B=true
C=true

```

图 9-37 运行结果

9.9.12 方法 public boolean isShutdown() 的使用

创建名称为 method11 的 Java 项目，运行类 Run1.java 代码如下：

```

package test.run;

import java.util.concurrent.ForkJoinPool;

public class Run1 {
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println("打印了! begin " + Thread.currentThread().getName());
                    Thread.sleep(1000);
                    System.out.println("打印了! end " + Thread.currentThread().getName());
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        };

        ForkJoinPool pool = new ForkJoinPool();
        pool.execute(runnable);
        System.out.println("A=" + pool.isShutdown());
        pool.shutdown();
        Thread.sleep(5000);
        System.out.println("B=" + pool.isShutdown());
    }
}

```

程序运行结果如图 9-38 所示。

任务成功被运行。

创建名称为 method12 的 Java 项目，运行类 Run1.java 代码如下：

```
package test.run;

import java.util.concurrent.ForkJoinPool;

public class Run1 {
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println("打印了! begin "
                        + Thread.currentThread().getName());
                    Thread.sleep(1000);
                    System.out.println("打印了!          end "
                        + Thread.currentThread().getName());
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        ForkJoinPool pool = new ForkJoinPool();
        pool.execute(runnable);
        System.out.println("A=" + pool.isShutdown());
        pool.shutdownNow();
        Thread.sleep(5000);
        System.out.println("B=" + pool.isShutdown());
    }
}
```

程序运行结果如图 9-39 所示。

```
A=false
打印了! begin ForkJoinPool-1-worker-1
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at test.run.Run1.run(Run1.java:13)
    at java.util.concurrent.ForkJoinTask$AdaptedRunnable.exec(ForkJoinTask.java:1265)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:334)
    at java.util.concurrent.ForkJoinWorkerThread.execTask(ForkJoinWorkerThread.java:604)
    at java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:646)
    at java.util.concurrent.ForkJoinPool.work(ForkJoinPool.java:646)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:398)
B=true
```

图 9-39 运行结果

任务成功被运行，由于 shutdownNow() 方法在源代码内部使用了 interrupt() 方法，所以 interrupt() 方法遇到 sleep() 抛出“java.lang.InterruptedException: sleep interrupted”异常。

```
(terminated) Run1 (2) [Java Application] C:\jdk1.7\bin
A=false
打印了! begin ForkJoinPool-1-worker-1
打印了!          end ForkJoinPool-1-worker-1
B=true
```

图 9-38 运行结果

如果使用 Callable 接口,则需要使用 Future 对象的 get() 方法获得异常。

9.9.13 方法 public boolean awaitTermination(long timeout, TimeUnit unit) 的使用

方法 awaitTermination(long timeout, TimeUnit unit) 的作用是等待池被销毁的最长时间,具有阻塞特性。

创建名称为 method13 的 Java 项目,创建类 MyRunnable1.java 代码如下:

```
package myrunnable;

public class MyRunnable1 implements Runnable {
    public void run() {
        try {
            System.out.println("begin " + Thread.currentThread().getName()
                + " " + System.currentTimeMillis());
            Thread.sleep(4000);
            System.out.println(" end " + Thread.currentThread().getName()
                + " " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

创建类 MyRunnable2.java 代码如下:

```
package myrunnable;

public class MyRunnable2 implements Runnable {
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

运行类 Test1.java 代码如下:

```
package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ForkJoinPool pool = new ForkJoinPool();
        pool.execute(myRunnable);
    }
}
```

```

        System.out.println("main begin ! " + System.currentTimeMillis());
        System.out.println(pool.awaitTermination(10, TimeUnit.SECONDS));
        System.out.println("main end ! " + System.currentTimeMillis());
        // 此实验说明 awaitTermination() 方法具有阻塞特性
    }
}

```

程序运行结果如图 9-40 所示。

返回值打印 false 代表任务池并没有被销毁。

运行类 Test2.java 代码如下：

```

package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ForkJoinPool pool = new ForkJoinPool();
        pool.execute(myRunnable);
        pool.shutdown();
        System.out.println("main begin ! " + System.currentTimeMillis());
        System.out.println(pool.awaitTermination(10, TimeUnit.SECONDS));
        System.out.println("main end ! " + System.currentTimeMillis());
        // 代码: awaitTermination(10, TimeUnit.SECONDS) 作用:
        // 最多等待 10 秒, 也就是阻塞 10 秒
    }
}

```

图 9-40 运行结果

程序运行结果如图 9-41 所示。

图 9-41 运行结果

日志 main begin 和 main end 之间的时差差 4 秒, 返回值打印 true, 代表任务池在 4 秒后被销毁, 所以 awaitTermination(long timeout, TimeUnit unit) 要结合 shutdown() 方法进行使用。

运行类 Test3.java 代码如下：

```

package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.TimeUnit;

```

```
import myrunnable.MyRunnable2;

public class Test3 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable2 myRunnable = new MyRunnable2();
        ForkJoinPool pool = new ForkJoinPool();
        pool.execute(myRunnable);
        pool.shutdown();
        System.out.println("A=" + pool.awaitTermination(1, TimeUnit.SECONDS)
            + " " + System.currentTimeMillis());
        System.out.println("B=" + pool.awaitTermination(1, TimeUnit.SECONDS)
            + " " + System.currentTimeMillis());
    }
}
```

程序运行结果如图 9-42 所示。

运行类 Test4.java 代码如下：

```
package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.TimeUnit;

import myrunnable.MyRunnable1;

public class Test4 {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable1 myRunnable = new MyRunnable1();
        ForkJoinPool pool = new ForkJoinPool();
        pool.execute(myRunnable);
        pool.shutdown();
        System.out.println(System.currentTimeMillis());
        System.out.println(pool.awaitTermination(Integer.MAX_VALUE,
            TimeUnit.SECONDS)
            + " " + System.currentTimeMillis() + " 全部任务执行完毕! ");
        System.out.println(System.currentTimeMillis());
    }
}
```

程序运行结果如图 9-43 所示。

```
(terminated) Test3 (3) [Java Appli...
A=false 1432176332656
B=false 1432176333656
```

图 9-42 运行结果

```
(terminated) Test4 [Java Application] C:\jdk1.7\bin\javaw.exe
1432176365859
begin ForkJoinPool-1-worker-1 1432176365859
end ForkJoinPool-1-worker-1 1432176369859
true 1432176369859 全部任务执行完毕!
1432176369859
```

图 9-43 运行结果

9.9.14 方法 public <T> T invoke(ForkJoinTask<T> task) 的使用

创建名称为 method14 的 Java 项目，创建类 MyRecursiveAction.java 代码如下：


```

package myaction;

import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction {

    @Override
    protected void compute() {
        System.out.println("ThreadName=" + Thread.currentThread().getName());
    }
}

```

创建类 MyRecursiveTask.java 代码如下：

```

package mytask;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<String> {

    @Override
    protected String compute() {
        return "我是返回值";
    }
}

```

运行类 Test1.java 代码如下：

```

package test;

import java.util.concurrent.ForkJoinPool;

import myaction.MyRecursiveAction;

public class Test1 {

    public static void main(String[] args) throws InterruptedException {
        MyRecursiveAction action = new MyRecursiveAction();
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(action);
    }
}

```

程序运行结果如图 9-44 所示。

运行类 Test2.java 代码如下：

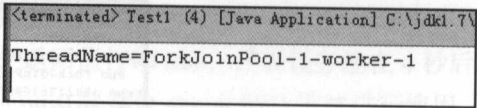
```

package test;

import java.util.concurrent.ForkJoinPool;

import mytask.MyRecursiveTask;

```



```

<terminated> Test1 (4) [Java Application] C:\jdk1.7\
ThreadName=ForkJoinPool-1-worker-1

```

图 9-44 运行结果

```

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyRecursiveTask action = new MyRecursiveTask();
        ForkJoinPool pool = new ForkJoinPool();
        String returnString = pool.invoke(action);
        System.out.println(returnString);
    }
}

```

<terminated> Test2 (4)
 我是返回值

图 9-45 运行结果

程序运行结果如图 9-45 所示。

方法 `execute(task)`、`submit(task)` 以及 `invoke(task)` 都可以在异步队列中执行任务，需要注意的是，方法 `invoke()` 是阻塞的，而它们在使用上的区别其实很简单，`execute(task)` 只执行任务，没有返回值，而 `submit(task)` 方法有返回值，返回值类型是 `ForkJoinTask`，想取得返回值时，需要使用 `ForkJoinTask` 对象的 `get()` 方法，而 `invoke(task)` 和 `submit(task)` 方法一样都具有返回值的功能，区别就是 `invoke(task)` 方法直接将返回值进行返回，而不是通过 `ForkJoinTask` 对象的 `get()` 方法。

这 3 个方法的声明如下：

```

pool.execute(task);
//public void execute(ForkJoinTask<?> task)
pool.submit(task);
//public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task)
//task.get();
pool.invoke(task);
//public <T> T invoke(ForkJoinTask<T> task)

```

9.9.15 监视 pool 池的状态

类提供了若干方法来监视任务池的状态：

- ❑ 方法 `getParallelism()`：获得并行的数量，与 CPU 的内核数有关；
- ❑ 方法 `getPoolSize()`：获得任务池的大小；
- ❑ 方法 `getQueuedSubmissionCount()`：取得已经提交但尚未被执行的任务数量；
- ❑ 方法 `hasQueuedSubmissions()`：判断队列中是否有未执行的任务；
- ❑ 方法 `getActiveThreadCount()`：获得活动的线程个数；
- ❑ 方法 `getQueuedTaskCount()`：获得任务的总个数；
- ❑ 方法 `getStealCount()`：获得偷窃的任务个数；
- ❑ 方法 `getRunningThreadCount()`：获得正在运行并且不在阻塞状态下的线程个数；
- ❑ 方法 `isQuiescent()`：判断任务池是否是静止未执行任务的状态。

为了验证这些方法的使用，创建项目 `method16`。

创建类 `MyRecursiveTask1.java` 代码如下：

```

package mytask;

import java.util.concurrent.RecursiveAction;

public class MyRecursiveTask1 extends RecursiveAction {
    protected void compute() {
        try {
            System.out.println("begin=" + Thread.currentThread().getName());
            Thread.sleep(1000);
            System.out.println("end=" + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

类 Run1_1.java 代码如下：

```

package test.run;

import java.util.concurrent.ForkJoinPool;

import mytask.MyRecursiveTask1;

public class Run1_1 {

    public static void main(String[] args) throws InterruptedException {
        MyRecursiveTask1 task1 = new MyRecursiveTask1();
        MyRecursiveTask1 task2 = new MyRecursiveTask1();
        MyRecursiveTask1 task3 = new MyRecursiveTask1();
        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(task1);
        pool.submit(task2);
        pool.submit(task3);
        System.out.println("并行数 getParallelism()=" + pool.getParallelism()
            + " 线程池大小 getPoolSize()=" + pool.getPoolSize());
        pool.shutdown();
        do {
        } while (!pool.isTerminated());
        System.out.println("main end!");
    }
}

```

程序运行结果如图 9-46 所示。

类 Run1_2.java 代码如下：

```

package test.run;

import java.util.concurrent.ForkJoinPool;

import mytask.MyRecursiveTask1;

```

```

begin=ForkJoinPool-1-worker-1
并行数getParallelism()=4 线程池大小getPoolSize()=4
begin=ForkJoinPool-1-worker-1
begin=ForkJoinPool-1-worker-2
end=ForkJoinPool-1-worker-3
end=ForkJoinPool-1-worker-1
end=ForkJoinPool-1-worker-2
main end!

```

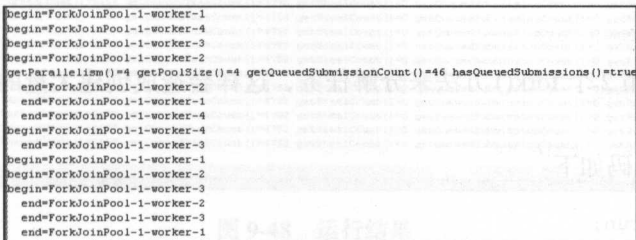
图 9-46 运行结果

```

public class Run1_2 {
    public static void main(String[] args) throws InterruptedException {
        MyRecursiveTask1 task1 = null;
        ForkJoinPool pool = new ForkJoinPool();
        for (int i = 0; i < 50; i++) {
            task1 = new MyRecursiveTask1();
            pool.submit(task1);
        }
        Thread.sleep(50);
        System.out.println("getParallelism()=" + pool.getParallelism()
            + " getPoolSize()=" + pool.getPoolSize()
            + " getQueuedSubmissionCount()="
            + pool.getQueuedSubmissionCount() + " hasQueuedSubmissions()="
            + pool.hasQueuedSubmissions());
        do {
            while (!task1.isDone());
        }
    }
}

```

程序运行结果如图 9-47 所示。



```

begin=ForkJoinPool-1-worker-1
begin=ForkJoinPool-1-worker-4
begin=ForkJoinPool-1-worker-3
begin=ForkJoinPool-1-worker-2
getParallelism()=4 getPoolSize()=4 getQueuedSubmissionCount()=46 hasQueuedSubmissions()=true
end=ForkJoinPool-1-worker-2
end=ForkJoinPool-1-worker-1
end=ForkJoinPool-1-worker-4
end=ForkJoinPool-1-worker-3
begin=ForkJoinPool-1-worker-1
begin=ForkJoinPool-1-worker-2
begin=ForkJoinPool-1-worker-3
end=ForkJoinPool-1-worker-2
end=ForkJoinPool-1-worker-3
end=ForkJoinPool-1-worker-1

```

图 9-47 运行结果

类 MyRecursiveTask2.java 代码如下：

```

package mytask;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask2 extends RecursiveTask<Integer> {
    private int beginPosition;
    private int endPosition;

    public MyRecursiveTask2(int beginPosition, int endPosition) {
        super();
        this.beginPosition = beginPosition;
        this.endPosition = endPosition;
    }

    protected Integer compute() {
        Integer sumValue = 0;
        if ((endPosition - beginPosition) > 2) {

```

```

    int middleNum = (endPosition + beginPosition) / 2;
    MyRecursiveTask2 leftTask = new MyRecursiveTask2(beginPosition,
        middleNum);
    MyRecursiveTask2 rightTask = new MyRecursiveTask2(middleNum + 1,
        endPosition);

    leftTask.fork();
    rightTask.fork();

    Integer leftValue = leftTask.join();
    Integer rightValue = rightTask.join();

    return leftValue + rightValue;
} else {
    int count = 0;
    for (int i = beginPosition; i <= endPosition; i++) {
        count = count + i;
    }
    return count;
}
}
}

```

前面的代码使用 2 个 fork() 方法来分解任务，这样会造成创建大量的线程对象，有碍于程序的运行效率。

类 Run2.java 代码如下：

```

package test.run;

import java.util.concurrent.ForkJoinPool;

import mytask.MyRecursiveTask2;

public class Run2 {

    public static void main(String[] args) throws InterruptedException {
        MyRecursiveTask2 task = new MyRecursiveTask2(1, 900000);
        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(task);
        do {
            System.out.println("getParallelism()=" + pool.getParallelism()
                + " getPoolSize()=" + pool.getPoolSize()
                + " getActiveThreadCount()=" + pool.getActiveThreadCount()
                + " getQueuedTaskCount()=" + pool.getQueuedTaskCount()
                + " getStealCount()=" + pool.getStealCount()
                + " getQueuedSubmissionCount()="
                + pool.getQueuedSubmissionCount()
                + " getRunningThreadCount()="
                + pool.getRunningThreadCount());
        } while (!task.isDone());
    }
}

```

程序运行结果如图 9-48 所示。

[illegible]

图 9-48 运行结果

从运行结果来看，的确是创建了非常多的线程对象。

类 MyRecursiveTask3.java 代码如下:

```
package mytask;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask3 extends RecursiveTask<Integer> {

    private int beginPosition;
    private int endPosition;

    public MyRecursiveTask3(int beginPosition, int endPosition) {
        super();
        this.beginPosition = beginPosition;
        this.endPosition = endPosition;
    }

    protected Integer compute() {
        Integer sumValue = 0;
        if ((endPosition - beginPosition) > 2) {
```



```

        int middleNum = (endPosition + beginPosition) / 2;
        MyRecursiveTask3 leftTask = new MyRecursiveTask3(beginPosition,
            middleNum);
        MyRecursiveTask3 rightTask = new MyRecursiveTask3(middleNum + 1,
            endPosition);

        this.invokeAll(leftTask, rightTask);

        return leftTask.join() + rightTask.join();
    } else {
        int count = 0;
        for (int i = beginPosition; i <= endPosition; i++) {
            count = count + i;
        }
        return count;
    }
}
}

```

此实验使用方法 `invokeAll()` 以优化的方式分解及运行任务，效率得到了保障。

类 `Run3.java` 代码如下：

```

package test.run;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;

import mytask.MyRecursiveTask3;

public class Run3 {

    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        MyRecursiveTask3 task = new MyRecursiveTask3(1, 900000);
        ForkJoinPool pool = new ForkJoinPool();
        pool.submit(task);
        do {
            System.out.println(" getActiveThreadCount()="
                + pool.getActiveThreadCount() + " getQueuedTaskCount()="
                + pool.getQueuedTaskCount() + " getStealCount()="
                + pool.getStealCount() + " getQueuedSubmissionCount()="
                + pool.getQueuedSubmissionCount()
                + " getRunningThreadCount()="
                + pool.getRunningThreadCount());
        } while (!task.isDone());
        System.out.println(task.get());
    }
}

```

程序运行结果如图 9-49 所示。

[illegible]

图 9-49 运行结果

创建项目 method17，类 MyRecursiveTask1.java 代码如下：

```
package mytask;

import java.util.concurrent.RecursiveAction;

public class MyRecursiveTask1 extends RecursiveAction {
    private int i = 0;

    protected void compute() {
        System.out.println("beginA=" + Thread.currentThread().getName());
        while (i < 100000) {
            String newString = new String();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();
            Math.random();

            System.out.println("    endA=" + Thread.currentThread().getName());
        }
    }
}
```

类 Run.java 代码如下：

```
package test.run;

import java.util.concurrent.ForkJoinPool;
import mytask.MyRecursiveTask1;

public class Run {

    public static void main(String[] args) throws InterruptedException {
        MyRecursiveTask1 task11 = new MyRecursiveTask1();

        ForkJoinPool pool = new ForkJoinPool();
        System.out.println(pool.isQuiescent());
        pool.submit(task11);
        Thread.sleep(1000);
        System.out.println(pool.isQuiescent());
    }
}
```

程序运行结果如图 9-50 所示。

```
true
beginA=ForkJoinPool-1-worker-1
false
```

图 9-50 运行结果

9.10 类 ForkJoinTask 对异常的处理

方法 isCompletedAbnormally() 判断任务是否出现异常，方法 isCompletedNormally() 判断任务是否正常执行完毕，方法 getException() 返回报错异常。

创建项目 forkjointask_method2，类 MyRecursiveTask.java 代码如下：

```
package mytask;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<Integer> {

    @Override
    protected Integer compute() {
        try {
            Thread.sleep(1000);
            Integer.parseInt("a");
        } catch (NumberFormatException e) {
            e.printStackTrace();
            throw e;
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return 100;
    }
}
```

需要注意的是,在 catch 语句块中需要抛出 `NumberFormatException` 异常, `MyRecursiveTask` 对象从而可以获得任务执行结果的情况。

运行类 `Test1.java` 代码如下:

```
package test;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

import mytask.MyRecursiveTask;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        MyRecursiveTask action1 = new MyRecursiveTask();
        ForkJoinPool pool = new ForkJoinPool();
        ForkJoinTask task = pool.submit(action1);
        System.out.println(task.isCompletedAbnormally() + " "
            + task.isCompletedNormally());
        Thread.sleep(2000);
        System.out.println(task.isCompletedAbnormally() + " "
            + task.isCompletedNormally());
        System.out.println(task.getException());
    }
}
```

程序运行结果如图 9-51 所示。

```
false false
java.lang.NumberFormatException: For input string: "a"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.parseInt(Integer.java:527)
    at mytask.MyRecursiveTask.compute(MyRecursiveTask.java:11)
    at mytask.MyRecursiveTask.compute(MyRecursiveTask.java:1)
    at java.util.concurrent.RecursiveTask.exec(RecursiveTask.java:93)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:334)
    at java.util.concurrent.ForkJoinWorkerThread.execTask(ForkJoinWorkerThread.java:604)
    at java.util.concurrent.ForkJoinPool.new(ForkJoinPool.java:784)
    at java.util.concurrent.ForkJoinPool.work(ForkJoinPool.java:646)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:398)
true false
java.lang.NumberFormatException
```

图 9-51 运行结果

9.11 本章总结

在本章主要介绍了 Fork-Join 分治编程类主要的 API, 需要细化掌握 `ForkJoinTask` 的 2 个常用子类的 fork 分解算法, 虽然分治编程可以有效地利用 CPU 资源, 但不要忘了分治编程而分治, 应该结合具体的业务场景来进行使用。

并发集合框架

在 JDK 中提供了丰富的集合框架工具，这些工具可以有效地对数据进行处理，虽然说本书是介绍并发相关的技术，但为了讲解集合框架的完整性，所以也一并将 List、Set、Map、Queue 等常用接口一起介绍，尽量让读者看完本章后对 JDK 的集合框架了解的更加全面与具体。

10.1 集合框架结构简要

Java 语言中的集合框架父接口是 Iterable，从这个接口向下一一进行继承，就可以得出完整的 Java 集合框架结构，但由于集合框架的继承与实现关系相当复杂，所以简化的接口结构如图 10-1 所示：

在结构图中可以发现，出现 3 个继承分支的结构是在 Collection 接口中，它是集合框架主要功能的抽象。

10.1.1 接口 Iterable

接口 Iterable 的主要作用就是迭代循环，接口结构声明如图 10-2 所示。

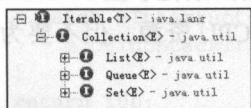


图 10-1 简化的集合框架接口结构

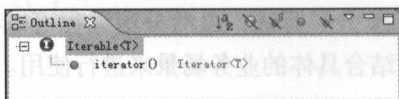


图 10-2 接口 Iterable 结构

接口结构非常简洁，仅有一个方法 iterator()，通过此方法返回 Iterator 对象，以进行循

环处理。

10.1.2 接口 Collection

接口 Collection 提供了集合框架最主要、最常用的操作，接口结构声明如图 10-3 所示。

接口内部提供的方法主要是针对数据的增删改查操作。

10.1.3 接口 List

接口 List 对 Collection 接口进行了扩展，允许根据索引位置操作数据，并且内容允许重复，接口结构声明如图 10-4 所示。

接口 List 最常用的非并发实现类就是 ArrayList，它是非线程安全的，该类实现了 List 接口，可以对数据以链表的形式进行组织，使数据有序排序。由于本书主要介绍的是并发集合，而 ArrayList 并不属于，另外也由于篇幅有限，所以针对 ArrayList 类的学习请查看源代码中名称为 ArrayListTest1 的 Java 项目。

类 ArrayList 并不是线程安全的，如果想使用线程安全的链表则可以使用 Vector 类，它的类信息如图 10-5 所示。

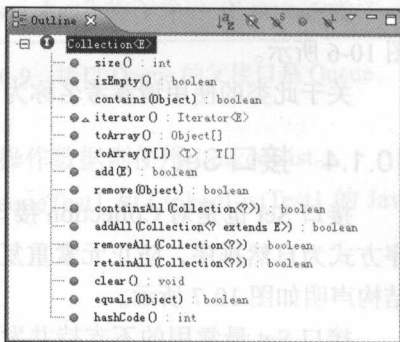


图 10-3 接口 Collection 结构

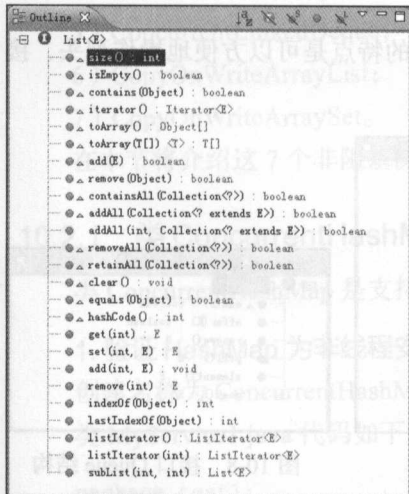


图 10-4 接口 List 结构

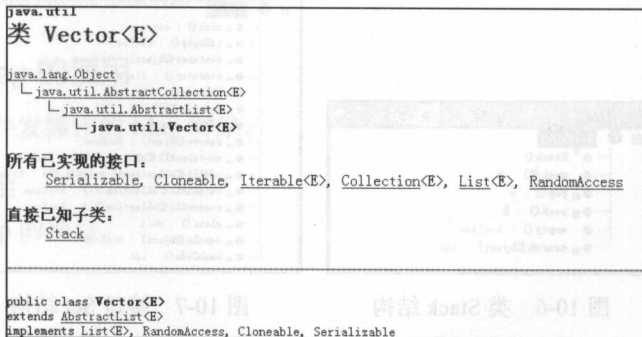


图 10-5 类 Vector 结构

类 Vector 是线程安全的，所以在多线程并发操作数据时也可以无误地处理集合中的数据。关于此类的使用请参考源代码中名称为 VectorTest1 的 Java 项目。需要说明一下，当多个线程分别调用该类的 iterator() 方法返回 Iterator 对象后，再调用 remove() 时会出现

ConcurrentModificationException 异常，也就是并不支持 Iterator 并发的删除，所以该类在功能上还是有缺陷。

类 Vector 有一个子类 Stack.java，它可以实现后进先出（LIFO）的对象堆栈，类结构如图 10-6 所示。

关于此类的使用请参考名称为 StackTest1 的项目。

10.1.4 接口 Set

接口 Set 也是对 Collection 接口进行了扩展，它具有的默认特点是内容不允许重复，排序方式为自然排序，防止元素重复的原理是元素需要重写 hashCode() 和 equals() 方法，接口结构声明如图 10-7 所示。

接口 Set 最常用的不支持并发的实现类就是 HashSet，关于此类的学习请查看源代码中名称为 HashSet 的 Java 项目。

HashSet 默认以无序的方式组织元素，而 LinkedHashSet 类可以有序的组织元素，此类的使用示例源代码在 LinkedHashSetTest1 的 Java 项目中。

接口 Set 还有另外一个实现类，名称为 TreeSet，它不仅实现了 Set 接口，而且还实现了 SortedSet 和 NavigableSet 接口，而 SortedSet 接口的父接口为 Set，SortedSet 和 NavigableSet 接口在功能上得到了扩展，比如可以获取 Set 中内容的子集，以比较范围进行获得子集，支持对表头与表尾的数据进行获取等，此类的实验代码在名称为 TreeSetTest1 的项目中。

10.1.5 接口 Queue

接口 Queue 是对 Collection 接口进行了扩展，它具有的特点是可以方便地操作列头，接口结构声明如图 10-8 所示。

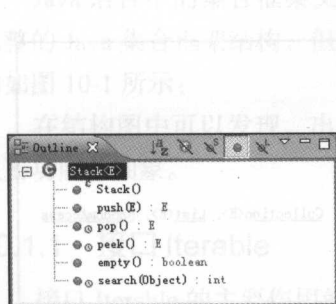


图 10-6 类 Stack 结构

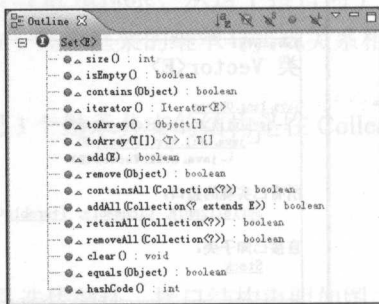


图 10-7 接口 Set 结构

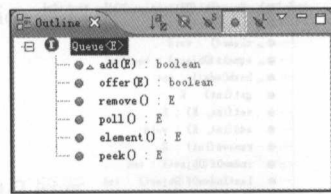


图 10-8 接口 Queue 结构

接口 Queue 的非并发实现类有 PriorityQueue，它是一个基于优先级的无界优先级队列，关于此类的使用请查看源代码中的 PriorityQueueTest1 项目。

10.1.6 接口 Deque

接口 Queue 可以支持对表头的操作，而接口 Deque 不仅支持对表头进行操作，而且还支

持对表尾进行操作，所以 Deque 的全称为“double ended queue（双端队列）”。

接口 Queue 和 Deque 之间有继承关系，如

图 10-9 所示。

```
193
194 public interface Deque<E> extends Queue<E> {
195     /**
```

图 10-9 接口 Deque 的父接口是 Queue

接口 Deque 的非并发实现类有 ArrayDeque

和 LinkedList，它们之间有一些区别，如果只想

实现从队列两端获取数据则使用 ArrayDeque，

如果想实现从队列两端获取数据时还可以根据索引的位置操作数据则使用 LinkedList。

关于这两个类的学习请查看源代码中名称为 ArrayDequeTest1 和 LinkedListTest1 的 Java 项目。

10.2 非阻塞队列

非阻塞队列的特色就是队列里面没有数据时，操作队列出现异常或返回 null，不具有等待 / 阻塞的特色。

在 JDK 的并发包中，常见的非阻塞队列有：

- 1) ConcurrentHashMap;
- 2) ConcurrentSkipListMap;
- 3) ConcurrentSkipListSet;
- 4) ConcurrentLinkedQueue;
- 5) ConcurrentLinkedDeque;
- 6) CopyOnWriteArrayList;
- 7) CopyOnWriteArraySet。

在本节将介绍这 7 个非阻塞队列的特点与使用。

10.2.1 类 ConcurrentHashMap 的使用

类 ConcurrentHashMap 是支持并发操作的 Map 对象。

1. 验证 HashMap 为非线程安全

创建名称为 ConcurrentHashMap 的项目。

类 MyService1.java 代码如下：

```
package test1;

import java.util.HashMap;

public class MyService1 {
    public HashMap map = new HashMap();
}
```

类 Thread1A.java 代码如下：

```
package test1;

public class Thread1A extends Thread {

    private MyService1 service;

    public Thread1A(MyService1 service) {
        super();
        this.service = service;
    }

    public void run() {
        for (int i = 0; i < 50000; i++) {
            service.map.put("ThreadA" + (i + 1), "ThreadA" + i + 1);
            System.out.println("ThreadA" + (i + 1));
        }
    }
}
```

类 Thread1B.java 代码如下：

```
package test1;

public class Thread1B extends Thread {

    private MyService1 service;

    public Thread1B(MyService1 service) {
        super();
        this.service = service;
    }

    public void run() {
        for (int i = 0; i < 50000; i++) {
            service.map.put("ThreadB" + (i + 1), "ThreadB" + i + 1);
            System.out.println("ThreadB" + (i + 1));
        }
    }
}
```

类 Test1_1.java 代码如下：

```
package test1;

public class Test1_1 {

    public static void main(String[] args) throws InterruptedException {
        MyService1 service = new MyService1();

        Thread1A a = new Thread1A(service);
    }
}
```

```

        a.start();
    }
}

```

程序运行结果如图 10-10 所示。

运行结果为 50000 个数据，是正确的，也就是单线程操作 HashMap 时是没有错误的，那么多线程呢？

类 Test1_2.java 代码如下：

```

package test1;

public class Test1_2 {

    public static void main(String[] args) throws InterruptedException {
        MyService1 service = new MyService1();

        Thread1A a = new Thread1A(service);
        Thread1B b = new Thread1B(service);

        a.start();
        b.start();
    }
}

```

程序运行结果如图 10-11 所示。

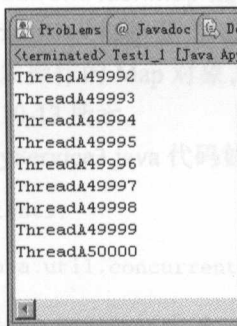


图 10-10 运行结果

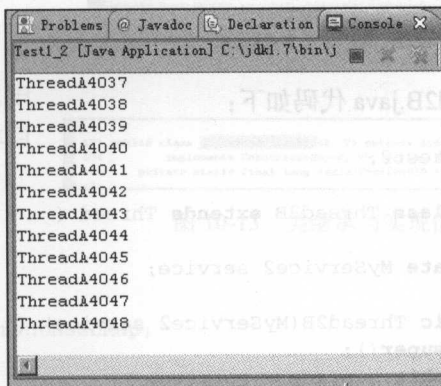


图 10-11 运行结果

程序运行后出现了“假死效果”，也就说明 HashMap 不能被多个线程所操作，也就是线程不安全。

2. 类 Hashtable 线程安全特性

上面的实验论证了 HashMap 不适合在多线程的情况下使用，如果想在多线程环境中使用 key-value 的数据结构，则可以使用 Hashtable 类。

类 MyService2.java 代码如下：

```
package test2;

import java.util.Hashtable;

public class MyService2 {
    public Hashtable map = new Hashtable();
}
```

类 Thread2A.java 代码如下：

```
package test2;

public class Thread2A extends Thread {

    private MyService2 service;

    public Thread2A(MyService2 service) {
        super();
        this.service = service;
    }

    public void run() {
        for (int i = 0; i < 50000; i++) {
            service.map.put("ThreadA" + (i + 1), "ThreadA" + i + 1);
            System.out.println("ThreadA" + (i + 1));
        }
    }
}
```

类 Thread2B.java 代码如下：

```
package test2;

public class Thread2B extends Thread {

    private MyService2 service;

    public Thread2B(MyService2 service) {
        super();
        this.service = service;
    }

    public void run() {
        for (int i = 0; i < 50000; i++) {
            service.map.put("ThreadB" + (i + 1), "ThreadB" + i + 1);
            System.out.println("ThreadB" + (i + 1));
        }
    }
}
```

类 Test2.java 代码如下：

```

package test2;

public class Test2 {

    public static void main(String[] args) throws InterruptedException {
        MyService2 service = new MyService2();

        Thread2A a = new Thread2A(service);
        Thread2B b = new Thread2B(service);

        a.start();
        b.start();

    }
}

```

程序运行结果如图 10-12 所示。

程序运行正确，每个线程添加 50000 个元素，说明 Hashtable 类在多线程的环境中不会出错，是线程安全的类。

当多个线程分别调用该类的 iterator() 方法返回 Iterator 对象后，再调用 remove() 时会出现 ConcurrentModificationException 异常，也就是并不支持 Iterator 并发的删除，建议使用并发集合框架提供的 ConcurrentHashMap 类。

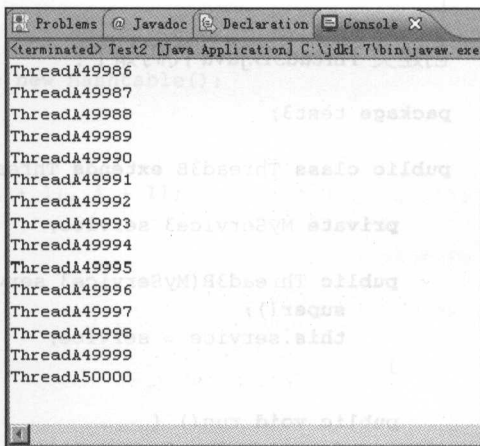


图 10-12 运行结果

3. 类 ConcurrentHashMap 的使用

类 ConcurrentHashMap 是 JDK 并发包中提供的支持并发操作的 Map 对象，类继承与实现信息如图 10-13 所示。

创建类 MyService3.java 代码如下：

```

package test3;

import java.util.concurrent.ConcurrentHashMap;

public class MyService3 {
    public ConcurrentHashMap map = new ConcurrentHashMap();
}

```

创建类 Thread3A.java 代码如下：

```

package test3;

public class Thread3A extends Thread {

    private MyService3 service;
}

```

```

102  */
103 public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
104     implements ConcurrentMap<K, V>, Serializable {
105     private static final long serialVersionUID = 7249069246763182397L;
106

```

图 10-13 类继承与实现信息


```

public Thread3A(MyService3 service) {
    super();
    this.service = service;
}

public void run() {
    for (int i = 0; i < 50000; i++) {
        service.map.put("ThreadA" + (i + 1), "ThreadA" + i + 1);
        System.out.println("ThreadA" + (i + 1));
    }
}
}
package test3;

```

创建类 Thread3B.java 代码如下：

```

package test3;

public class Thread3B extends Thread {
    private MyService3 service;

    public Thread3B(MyService3 service) {
        super();
        this.service = service;
    }

    public void run() {
        for (int i = 0; i < 50000; i++) {
            service.map.put("ThreadB" + (i + 1), "ThreadB" + i + 1);
            System.out.println("ThreadB" + (i + 1));
        }
    }
}

```

创建类 Test3.java 代码如下：

```

package test3;

public class Test3 {
    public static void main(String[] args) {
        MyService3 service = new MyService3();

        Thread3A a = new Thread3A(service);
        Thread3B b = new Thread3B(service);

        a.start();
        b.start();
    }
}

```

程序运行结果如图 10-14 所示。

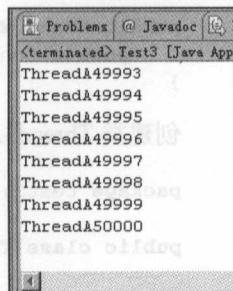


图 10-14 运行结果

此运行结果说明类 `ConcurrentHashMap` 支持在多线程的环境中使用。

`Hashtable` 和 `ConcurrentHashMap` 都支持并发操作，它们之间有什么差异呢？其实主要的差异就是 `Hashtable` 不支持在循环中 `remove()` 元素，下面来做一个实验。

4. 类 `Hashtable` 迭代中与有异常的测试

创建类 `MyService4.java` 代码如下：

```
package test4;

import java.util.Hashtable;

public class MyService4 {

    public static Hashtable hashtable = new Hashtable();

    public MyService4() {
        for (int i = 0; i < 5; i++) {
            hashtable.put("String" + (i + 1), i + 1);
        }
    }
}
```

创建类 `Thread4A.java` 代码如下：

```
package test4;

import java.util.Iterator;

public class Thread4A extends Thread {

    private MyService4 myService;

    public Thread4A(MyService4 myService) {
        super();
        this.myService = myService;
    }

    @Override
    public void run() {
        try {
            Iterator iterator = myService.hashtable.keySet().iterator();
            while (iterator.hasNext()) {
                System.out.println(iterator.next());
                Thread.sleep(3000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

创建类 Thread4B.java 代码如下：

```
package test4;

public class Thread4B extends Thread {

    private MyService4 myService;

    public Thread4B(MyService4 myService) {
        super();
        this.myService = myService;
    }

    @Override
    public void run() {
        myService.hashtable.put("z", "zValue");
    }
}
```

创建类 Test4.java 代码如下：

```
package test4;

public class Test4 {

    public static void main(String[] args) throws InterruptedException {
        try {
            MyService4 myService = new MyService4();

            Thread4A a = new Thread4A(myService);
            a.start();

            Thread.sleep(1000);

            Thread4B b = new Thread4B(myService);
            b.start();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
String5
Exception in thread "Thread-0" java.util.ConcurrentModificationException
    at java.util.Hashtable$Enumerator.next(Hashtable.java:1167)
    at test4.Thread4A.run(Thread4A.java:19)
```

图 10-15 出现异常

程序运行效果如图 10-15 所示。

说明 Hashtable 在获得了 Iterator 对象后，不允许更改其结构，否则出现 java.util.ConcurrentModificationException 异常。

但 ConcurrentHashMap 却支持这个功能。

5. 类 ConcurrentHashMap 迭代中与无异常的测试

创建类 MyService5.java 代码如下：

```

package test5;

import java.util.concurrent.ConcurrentHashMap;

public class MyService5 {

    public ConcurrentHashMap map = new ConcurrentHashMap();

    public MyService5() {
        for (int i = 0; i < 5; i++) {
            map.put("key" + (i + 1), "value" + (i + 1));
        }
    }
}

```

创建类 Thread5A.java 代码如下:

```

package test5;

import java.util.Iterator;

public class Thread5A extends Thread {

    private MyService5 myService;

    public Thread5A(MyService5 myService) {
        super();
        this.myService = myService;
    }

    @Override
    public void run() {
        try {
            Iterator iterator = myService.map.keySet().iterator();
            while (iterator.hasNext()) {
                System.out.println(iterator.next());
                Thread.sleep(3000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

创建类 Thread5B.java 代码如下:

```

package test5;

public class Thread5B extends Thread {

    private MyService5 myService;

    public Thread5B(MyService5 myService) {

```

创建类 `super()`；Java 代码如下：

```

    this.myService = myService;
}

```

```

@Override
public void run() {
    myService.map.put("z", "zValue");
}
}

```

创建类 `Test5.java` 代码如下：

```

package test5;

public class Test5 {

    public static void main(String[] args) throws InterruptedException {
        try {
            MyService5 myService = new MyService5();
            Thread5A a = new Thread5A(myService);
            a.start();

            Thread.sleep(1000);

            Thread5B b = new Thread5B(myService);
            b.start();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 成功但不支持排序
        // LinkedHashMap 虽然能支持顺序性，但又不支持并发
    }
}

```

key2
key4
key1
key5
key3
z

图 10-16 结构更改未出现异常

程序运行结果如图 10-16 所示。

运行结果是成功的，但 `ConcurrentHashMap` 不支持排序，虽然 `LinkedHashMap` 支持 key 的顺序性，但又不支持并发，那么如果出现这种既要求并发安全性，而又要求排序的情况就可以使用类 `ConcurrentSkipListMap`。

10.2.2 类 `ConcurrentSkipListMap` 的使用

类 `ConcurrentSkipListMap` 支持排序。

创建名称为 `ConcurrentSkipListMap` 的项目，创建类 `Userinfo.java` 代码如下：

```

package test1;

public class Userinfo implements Comparable<Userinfo> {

```

```

private int id;
private String username;

public Userinfo() {
    super();
}

public Userinfo(int id, String username) {
    super();
    this.id = id;
    this.username = username;
}

```

图 10-17 运行结果

支持排序而且不允许重复的，可调用 `java.util.concurrent.ConcurrentSkipListSet` 类实现。

创建名为 `ConcurrentSkipListSet` 的项目，创建类 `MyService1` 的代码如下：

```

package test1;

import java.util.concurrent.ConcurrentSkipListMap;

public class MyService1 {

    private ConcurrentSkipListMap<Userinfo> map;

    public MyService1() {
        map = new ConcurrentSkipListMap<>();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public int compareTo(Userinfo u) {
        if (this.getId() < u.getId()) {
            return -1;
        }
        if (this.getId() > u.getId()) {
            return 1;
        }
        return 0;
    }
}

```

创建类 `MyService1.java` 代码如下：

```

package test1;

import java.util.concurrent.ConcurrentSkipListMap;

public class MyService1 {

    public ConcurrentSkipListMap map = new ConcurrentSkipListMap<>();

    public MyService1() {

```



```

        Userinfo userinfo1 = new Userinfo(1, "username1");
        Userinfo userinfo3 = new Userinfo(3, "username3");
        Userinfo userinfo5 = new Userinfo(5, "username5");
        Userinfo userinfo4 = new Userinfo(4, "username4");
        Userinfo userinfo2 = new Userinfo(2, "username2");
        map.put(userinfo1, "value1");
        map.put(userinfo3, "value3");
        map.put(userinfo5, "value5");
        map.put(userinfo4, "value4");
        map.put(userinfo2, "value2");
    }
}

```

创建类 ThreadA.java 代码如下：

```

package test1;

import java.util.Map.Entry;

public class ThreadA extends Thread {

    private MyService1 service;

    public ThreadA(MyService1 service) {
        super();
        this.service = service;
    }

    public void run() {
        try {
            while (!service.map.isEmpty()) {
                Entry entry = service.map.pollFirstEntry();
                Userinfo userinfo = (Userinfo) entry.getKey();
                System.out.println(userinfo.getId() + " "
                    + userinfo.getUsername() + " " + entry.getValue());
                Thread.sleep((long) (Math.random() * 1000));
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

创建类 Test1.java 代码如下：

```

package test1;

import java.util.concurrent.ConcurrentSkipListMap;

public class Test1 {

    public static void main(String[] args) {
        MyService1 service = new MyService1();
        ThreadA a = new ThreadA(service);
    }
}

```

```

ThreadA b = new ThreadA(service);

a.start();
b.start();

}
}

```

程序运行结果如图 10-17 所示。

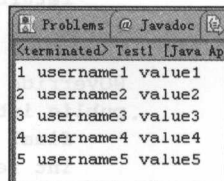


图 10-17 运行结果

10.2.3 类 ConcurrentSkipListSet 的使用

类 ConcurrentSkipListSet 支持排序而且不允许重复的元素。

创建名称为 ConcurrentSkipListSet 的项目，创建类 Userinfo.java 代码如下：

```

package test1;

public class Userinfo implements Comparable<Userinfo> {

    private int id;
    private String username;

    public Userinfo() {
        super();
    }

    public Userinfo(int id, String username) {
        super();
        this.id = id;
        this.username = username;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public int compareTo(Userinfo u) {
        if (this.getId() < u.getId()) {
            return -1;
        }
    }
}

```

```

        if (this.getId() > u.getId()) {
            return 1;
        }
        return 0;
    }

```

```

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    result = prime * result
        + ((username == null) ? 0 : username.hashCode());
    return result;
}

```

```

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Userinfo other = (Userinfo) obj;
    if (id != other.id)
        return false;
    if (username == null) {
        if (other.username != null)
            return false;
    } else if (!username.equals(other.username))
        return false;
    return true;
}
}

```

创建类 MyService1.java 代码如下：

```

package test1;

import java.util.concurrent.ConcurrentSkipListSet;

public class MyService1 {

    public ConcurrentSkipListSet map = new ConcurrentSkipListSet();

    public MyService1() {
        Userinfo userinfo1 = new Userinfo(1, "username1");
        Userinfo userinfo3 = new Userinfo(3, "username3");
        Userinfo userinfo5 = new Userinfo(5, "username5");
        Userinfo userinfo4 = new Userinfo(4, "username4");
        Userinfo userinfo44 = new Userinfo(4, "username4");
    }
}

```

```

Userinfo userinfo2 = new Userinfo(2, "username2");
map.add(userinfo1);
map.add(userinfo3);
map.add(userinfo5);
map.add(userinfo4);
map.add(userinfo44);
map.add(userinfo2);
}
}

```

创建类 ThreadA.java 代码如下:

```

package test1;

public class ThreadA extends Thread {

    private MyService1 service;

    public ThreadA(MyService1 service) {
        super();
        this.service = service;
    }

    public void run() {
        try {
            while (!service.map.isEmpty()) {
                Userinfo userinfo = (Userinfo) service.map.pollFirst();
                System.out.println(userinfo.getId() + " " +
                    userinfo.getUsername());
                Thread.sleep((long) (Math.random() * 1000));
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

创建类 Test1.java 代码如下:

```

package test1;

public class Test1 {

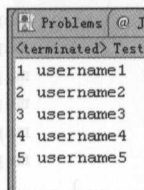
    public static void main(String[] args) {
        MyService1 service = new MyService1();
        ThreadA a = new ThreadA(service);
        ThreadA b = new ThreadA(service);

        a.start();
        b.start();
    }
}

```

程序运行结果如图 10-18 所示。

从运行结果来看，成功排序，并且不支持重复数据。



Problems @ J	
<terminated>	Test
1	username1
2	username2
3	username3
4	username4
5	username5

10.2.4 类 ConcurrentLinkedQueue 的使用

类 ConcurrentLinkedQueue 提供了并发环境的队列操作。

方法 poll() 当没有获得数据时返回为 null，如果有数据时则移除表头，并将表头进行返回。

图 10-18 运行结果

方法 element() 当没有获得数据时出现 NoSuchElementException 异常，如果有数据时则返回表头项。

方法 peek() 当没有获得数据时返回为 null，如果有数据时则不移除表头，并将表头进行返回。

创建名称为 ConcurrentLinkedQueue 的项目，创建类 MyService1.java 代码如下：

```
package myservice;

import java.util.concurrent.ConcurrentLinkedQueue;

public class MyService1 {
    public ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();
}
```

类 ThreadA.java 代码如下：

```
package test1;

import myservice.MyService1;

public class ThreadA extends Thread {

    private MyService1 service;

    public ThreadA(MyService1 service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            service.queue.add("threadA" + (i + 1));
        }
    }
}
```

类 ThreadB.java 代码如下：

```

package test1;

import myservice.MyService1;

public class ThreadB extends Thread {

    private MyService1 service;

    public ThreadB(MyService1 service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            service.queue.add("threadB" + (i + 1));
        }
    }
}

```

类 Test1.java 代码如下:

```

package test1;

import myservice.MyService1;

public class Test1 {

    public static void main(String[] args) {
        try {
            MyService1 service = new MyService1();
            ThreadA a = new ThreadA(service);
            ThreadB b = new ThreadB(service);

            a.start();
            b.start();
            a.join();
            b.join();

            System.out.println(service.queue.size());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 10-19 所示。

创建类代码如下:


```
package test2;

import myservice.MyService1;

public class Test2_1 {
    public static void main(String[] args) {
        MyService1 service = new MyService1();
        System.out.println(service.queue.poll());
    }
}
```

程序运行结果如图 10-20 所示。

创建类 Test2_2.java 代码如下：

```
package test2;

import myservice.MyService1;

public class Test2_2 {

    public static void main(String[] args) {
        MyService1 service = new MyService1();
        service.queue.add("a");
        service.queue.add("b");
        service.queue.add("c");
        System.out.println(service.queue.poll());
    }
}
```

程序运行结果如图 10-21 所示。

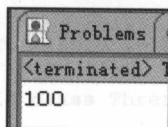


图 10-19 支持并发环境下
的添加元素

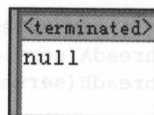


图 10-20 无数据时返回
为 null

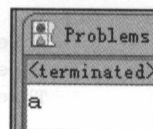


图 10-21 支持对列头
进行操作

10.2.5 类 ConcurrentLinkedDeque 的使用

类 ConcurrentLinkedQueue 仅支持对列头进行操作，而 ConcurrentLinkedDeque 支持对列头列尾双向操作。

创建名称为 ConcurrentLinkedDeque 的项目，创建类 MyService.java 代码如下：

```
package myservice;

import java.util.concurrent.ConcurrentLinkedDeque;

public class MyService {
    public ConcurrentLinkedDeque queue = new ConcurrentLinkedDeque();
}
```

```

public MyService() {
    for (int i = 0; i < 10000; i++) {
        queue.add("string" + (i + 1));
    }
}
}

```

创建类 ThreadA.java 代码如下:

```

package extthread;

import myservice.MyService;

public class ThreadA extends Thread {

    private MyService service;

    public ThreadA(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        while (!service.queue.isEmpty()) {
            service.queue.pollFirst();
            System.out.println(service.queue.size());
        }
    }
}

```

创建类 ThreadB.java 代码如下:

```

package extthread;

import myservice.MyService;

public class ThreadB extends Thread {

    private MyService service;

    public ThreadB(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        while (!service.queue.isEmpty()) {
            service.queue.pollLast();
            System.out.println(service.queue.size());
        }
    }
}

```

创建类 Test.java 代码如下：

```
package test1;

import myservice.MyService;
import extthread.ThreadA;
import extthread.ThreadB;

public class Test {

    public static void main(String[] args) {
        try {
            MyService service = new MyService();
            ThreadA a = new ThreadA(service);
            ThreadB b = new ThreadB(service);

            a.start();
            b.start();
            a.join();
            b.join();

            System.out.println(service.queue.size());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 10-22 所示。

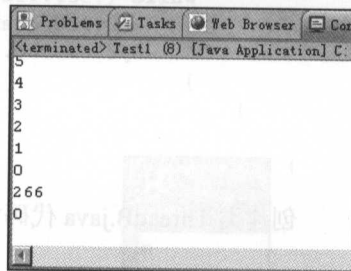


图 10-22 成功操作列头列尾

10.2.6 类 CopyOnWriteArrayList 的使用

前面实验介绍过，ArrayList 为非线程安全的，如果想在并发中实现线程安全，则可以使用 CopyOnWriteArrayList 类。

创建名称为 CopyOnWriteArrayList 的项目，类 MyServiceA.java 代码如下：

```
package test1;

import java.util.ArrayList;
import java.util.List;

public class MyServiceA {
    public static List list = new ArrayList();
}
```

创建类 ThreadA.java 代码如下：

```

package test1;

public class ThreadA extends Thread {

    private MyServiceA service;

    public ThreadA(MyServiceA service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            service.list.add("anyString");
        }
    }
}

```

创建类 Test1.java 代码如下:

```

package test1;

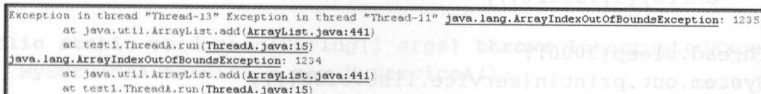
public class Test1 {

    public static void main(String[] args) throws InterruptedException {
        MyServiceA service = new MyServiceA();

        ThreadA[] aArray = new ThreadA[100];
        for (int i = 0; i < aArray.length; i++) {
            aArray[i] = new ThreadA(service);
        }
        for (int i = 0; i < aArray.length; i++) {
            aArray[i].start();
        }
        Thread.sleep(3000);
        System.out.println(service.list.size());
    }
}

```

程序运行后,按一定的概率会出现异常,如图 10-23 所示。



```

Exception in thread "Thread-13" Exception in thread "Thread-11" java.lang.ArrayIndexOutOfBoundsException: 1235
    at java.util.ArrayList.add(ArrayList.java:441)
    at test1.ThreadA.run(ThreadA.java:15)
    at java.lang.Thread.run(Thread.java:745)
java.lang.ArrayIndexOutOfBoundsException: 1234
    at java.util.ArrayList.add(ArrayList.java:441)
    at test1.ThreadA.run(ThreadA.java:15)

```

图 10-23 类 ArrayList 不安全

在这种情况下可以使用类 CopyOnWriteArrayList 作为替代。

创建类 MyServiceB.java 代码如下:

```
package test2;

import java.util.concurrent.CopyOnWriteArrayList;

public class MyServiceB {
    public static CopyOnWriteArrayList list = new CopyOnWriteArrayList();
}
```

创建类 ThreadB.java 代码如下：

```
package test2;

public class ThreadB extends Thread {

    private MyServiceB service;

    public ThreadB(MyServiceB service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            service.list.add("anyString");
        }
    }
}
```

创建类 Test2.java 代码如下：

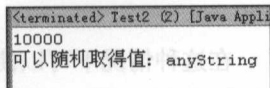
```
package test2;

public class Test2 {

    public static void main(String[] args) throws InterruptedException {
        MyServiceB service = new MyServiceB();

        ThreadB[] aArray = new ThreadB[100];
        for (int i = 0; i < aArray.length; i++) {
            aArray[i] = new ThreadB(service);
        }
        for (int i = 0; i < aArray.length; i++) {
            aArray[i].start();
        }
        Thread.sleep(3000);
        System.out.println(service.list.size());
        System.out.println("可以随机取得值: " + service.list.get(5));
    }
}
```

程序运行结果如图 10-24 所示。



```
<terminated> Test2 (2) [Java Appli...
10000
可以随机取得值: anyString
```

图 10-24 运行结果是正确的

10.2.7 类 CopyOnWriteArraySet 的使用

与 CopyOnWriteArrayList 配套的还有一个类叫做 CopyOnWriteArraySet，它也可以解决多线程的情况下 HashSet 不安全的问题。

创建名称为 CopyOnWriteArraySet 的项目，类 MyServiceA.java 代码如下：

```
package test1;

import java.util.HashSet;
import java.util.Set;

public class MyServiceA {
    public static Set set = new HashSet();
}
```

创建类 ThreadA.java 代码如下：

```
package test1;

public class ThreadA extends Thread {

    private MyServiceA service;

    public ThreadA(MyServiceA service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            service.set.add(Thread.currentThread().getName() + "anyString"
                + (i + 1));
        }
    }
}
```

创建类 Test1.java 代码如下：

```
package test1;

public class Test1 {

    public static void main(String[] args) throws InterruptedException {
        MyServiceA service = new MyServiceA();

        ThreadA[] aArray = new ThreadA[100];
        for (int i = 0; i < aArray.length; i++) {
            aArray[i] = new ThreadA(service);
        }
        for (int i = 0; i < aArray.length; i++) {
            aArray[i].start();
        }
    }
}
```

```

    }
    Thread.sleep(3000);
    System.out.println(service.set.size());
}
}

```

程序运行结果如图 10-25 所示。

在多线程的环境下使用 HashSet 是不安全的，可以使用 CopyOnWriteArraySet 类作为替代。

创建类 MyServiceB.java 代码如下：

```

package test2;

import java.util.concurrent.CopyOnWriteArraySet;

public class MyServiceB {
    public static CopyOnWriteArraySet set = new CopyOnWriteArraySet();
}

```

创建类 ThreadB.java 代码如下：

```

package test2;

public class ThreadB extends Thread {
    private MyServiceB service;

    public ThreadB(MyServiceB service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            service.set.add(Thread.currentThread().getName() + "anyString"
                + (i + 1));
        }
    }
}

```

创建类 Test2.java 代码如下：

```

package test2;

public class Test2 {
    public static void main(String[] args) throws InterruptedException {
        MyServiceB service = new MyServiceB();

        ThreadB[] aArray = new ThreadB[100];
        for (int i = 0; i < aArray.length; i++) {

```



```

        aArray[i] = new ThreadB(service);
    }
    for (int i = 0; i < aArray.length; i++) {
        aArray[i].start();
    }
    Thread.sleep(3000);
    System.out.println(service.set.size());
}
}

```

程序运行结果如图 10-26 所示。

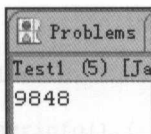


图 10-25 并没有达到预期值 10000

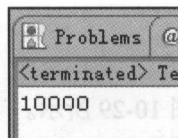


图 10-26 线程安全的运行结果

10.3 阻塞队列

在 JDK 中提供了若干集合工具类都具有阻塞特性，所谓的阻塞队列 `BlockingQueue`，其实就是如果 `BlockQueue` 是空的，从 `BlockingQueue` 取东西的操作将会被阻塞进入等待状态，直到 `BlockingQueue` 添加进了元素才会被唤醒。同样，如果 `BlockingQueue` 是满的，也就是没有空余空间时，试图往队列中存放元素的操作也会被阻塞进入等待状态，直到 `BlockingQueue` 里有剩余空间才会被唤醒继续操作。

```

75 public class ArrayBlockingQueue<E> extends AbstractQueue<E>
76     implements BlockingQueue<E>, java.io.Serializable {

```

图 10-27 类 `ArrayBlockingQueue` 继承信息

10.3.1 类 `ArrayBlockingQueue` 的使用

类 `ArrayBlockingQueue` 提供一种有界阻塞队列的功能，类继承信息如图 10-27 所示。

其中 `BlockingQueue` 阻塞队列接口结构如图 10-28 所示。

创建名称为 `ArrayBlockingQueueTest1` 的项目。
类 `put.java` 代码如下：

```

package test;

import java.util.concurrent.ArrayBlockingQueue;

public class put {

    public static void main(String[] args) {

```

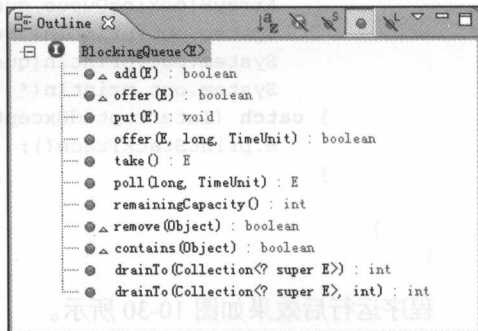


图 10-28 接口 `BlockingQueue` 结构

```

try {
    ArrayBlockingQueue queue = new ArrayBlockingQueue(3);
    queue.put("a1");
    queue.put("a2");
    queue.put("a3");
    System.out.println(queue.size());
    System.out.println(System.currentTimeMillis());
    queue.put("a4");
    System.out.println(System.currentTimeMillis());
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
}

```

程序运行结果如图 10-29 所示。

出现阻塞的原因是代码 `new ArrayBlockingQueue(3)` 只创建了容纳 3 个元素的集合，所以当添加到第 4 个时添加不了，就呈阻塞状态，等待某一时间有空余空间时再继续添加。

方法 `put` 是存放数据，如果没有空余的空间存放数据时，则呈阻塞状态。其实在获取元素时如果没有元素可获取，也会呈阻塞状态。

创建类 `take.java` 代码如下：

```

package test;

import java.util.concurrent.ArrayBlockingQueue;

public class take {

    public static void main(String[] args) {
        try {
            ArrayBlockingQueue queue = new ArrayBlockingQueue(3);
            System.out.println("begin " + System.currentTimeMillis());
            System.out.println(queue.take());
            System.out.println("end " + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行后效果如图 10-30 所示。

完整的方法功能演示请查看源代码中名称为 `ArrayBlockingQueueTest1` 的项目。

10.3.2 类 `PriorityBlockingQueue` 的使用

类 `PriorityBlockingQueue` 支持在并发情况下的优先级队列，其类继承信息如图 10-31 所示。

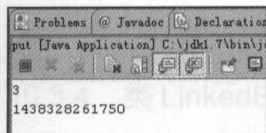


图 10-29 阻塞了

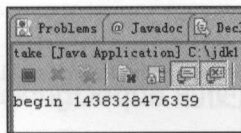
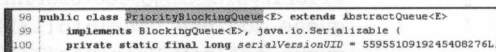
图 10-30 等待数据呈
阻塞状态

图 10-31 类 PriorityBlockingQueue 的继承信息

创建名称为 PriorityBlockingQueueTest 的项目，类 Userinfo.java 代码如下：

```
package entity;
```

```
public class Userinfo implements Comparable<Userinfo> {
```

```
    private int id;
```

```
    public Userinfo() {
        super();
    }
```

```
    public Userinfo(int id) {
        super();
        this.id = id;
    }
```

```
    public int getId() {
        return id;
    }
```

```
    public void setId(int id) {
        this.id = id;
    }
```

```
@Override
```

```
    public int compareTo(Userinfo o) {
        if (this.id < o.getId()) {
            return -1;
        }
        if (this.id > o.getId()) {
            return 1;
        }
        return 0;
    }
```

```
}
```

类 Test1.java 代码如下：

```
package test;
```

```
import java.util.concurrent.PriorityBlockingQueue;
```

```
import entity.Userinfo;
```

```

public class Test1 {
    public static void main(String[] args) {
        PriorityQueue<Userinfo> queue = new PriorityQueue<Userinfo>();
        queue.add(new Userinfo(12));
        queue.add(new Userinfo(13478));
        queue.add(new Userinfo(1569));
        queue.add(new Userinfo(1346));
        queue.add(new Userinfo(1762));
        queue.add(new Userinfo(1876876));

        System.out.println(queue.poll().getId());
        System.out.println(queue.poll().getId());
        System.out.println(queue.poll().getId());
        System.out.println(queue.poll().getId());
        System.out.println(queue.poll().getId());
        System.out.println(queue.poll().getId());
        System.out.println(queue.poll());
    }
}

```

程序运行后的效果如图 10-32 所示。

创建类 Test2.java 代码如下：

```

package test;

import java.util.concurrent.PriorityBlockingQueue;
import entity.Userinfo;

public class Test2 {
    public static void main(String[] args) {
        try {
            PriorityQueue<Userinfo> queue = new PriorityQueue<Userinfo>();
            System.out.println("begin");
            System.out.println(queue.take());
            System.out.println("end");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果如图 10-33 所示。

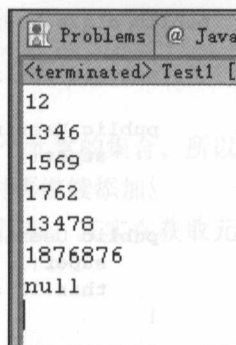


图 10-32 成功排序打印

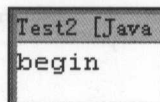


图 10-33 出现阻塞

10.3.3 类 LinkedBlockingQueue 的使用

类 LinkedBlockingQueue 和 ArrayBlockingQueue 在功能上大体一样，只不过 ArrayBlockingQueue 是有界的，而 LinkedBlockingQueue 是无界的，当然 LinkedBlockingQueue 类也可以定义成是有界的，但它们两者都有阻塞特性。API 的功能与其他类在使用上大体是一致的，更

加具体的 API 演示请查看源代码中的 `LinkedBlockingQueueTest1` 项目。

10.3.4 类 `LinkedBlockingDeque` 的使用

类 `LinkedBlockingQueue` 和 `LinkedBlockingDeque` 在功能上有差异, 类 `LinkedBlockingQueue` 只支持对列头的操作, 而 `LinkedBlockingDeque` 类提供对双端结点的操作, 两者都具有阻塞特性。API 的功能与其他类在使用上大体是一致的, 更加具体的 API 使用演示请查看源代码中的 `LinkedBlockingDequeTest1` 项目。

10.3.5 类 `SynchronousQueue` 的使用

类 `SynchronousQueue` 为异步队列。

一种阻塞队列, 其中每个插入操作必须等待另一个线程的对应移除操作, 反之亦然。同步队列没有任何内部容量, 甚至连一个队列的容量都没有。不能在同步队列上进行 `peek`, 因为仅在试图要移除元素时, 该元素才存在; 除非另一个线程试图移除某个元素, 否则也不能 (使用任何方法) 插入元素; 也不能迭代队列, 因为其中没有元素可用于迭代。

类 `SynchronousQueue` 经常在多个线程之间传输数据时使用。

创建名称为 `SynchronousQueueTest1` 的项目, 类 `MyService.java` 代码如下:

```
package service;

import java.util.concurrent.SynchronousQueue;

public class MyService {

    public static SynchronousQueue queue = new SynchronousQueue();

    public void putMethod() {
        try {
            String putString = "anyString" + Math.random();
            System.out.println(" put=" + putString);
            queue.put(putString);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void takeMethod() {
        try {
            System.out.println("take=" + queue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 `ThreadPut.java` 代码如下:

```

package extthread;

import service.MyService;

public class ThreadPut extends Thread {

    private MyService service;

    public ThreadPut(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            service.putMethod();
        }
    }
}

```

类 ThreadTake.java 代码如下：

```

package extthread;

import service.MyService;

public class ThreadTake extends Thread {

    private MyService service;

    public ThreadTake(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            service.takeMethod();
        }
    }
}

```

类 Test1.java 代码如下：

```

package test;

import java.util.concurrent.SynchronousQueue;

public class Test1 {

    public static void main(String[] args) {
        try {
            SynchronousQueue queue = new SynchronousQueue();

```

```

        System.out.println("step1");
        queue.put("anyString");
        System.out.println("step2");
        System.out.println(queue.take());
        System.out.println("step3");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

程序运行结果如图 10-34 所示。

阻塞的原因是数据并没有被其他线程移走，所以一直呈阻塞状态，程序不能继续向下运行。

类 Test2.java 代码如下：

```

package test;

import service.MyService;
import extthread.ThreadPut;
import extthread.ThreadTake;

public class Test2 {

    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();

        ThreadPut threadPut = new ThreadPut(service);
        ThreadTake threadTake = new ThreadTake(service);

        threadTake.start();
        Thread.sleep(2000);

        threadPut.start();
    }
}

```

程序运行结果如图 10-35 所示。

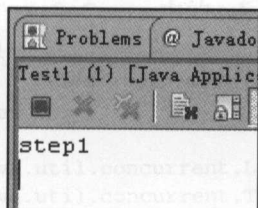


图 10-34 阻塞了

```

put=anyString0.3530550444590701
take=anyString0.3530550444590701
put=anyString0.6353832977175975
take=anyString0.6353832977175975
put=anyString0.9008682971514572
take=anyString0.9008682971514572
put=anyString0.9747861989807773
take=anyString0.9747861989807773
put=anyString0.9756930136000809
take=anyString0.9756930136000809
put=anyString0.6708089775046417
take=anyString0.6708089775046417
put=anyString0.9416945166664107
take=anyString0.9416945166664107
put=anyString0.5999442422522446
take=anyString0.5999442422522446
put=anyString0.2021813158678225
take=anyString0.2021813158678225
put=anyString0.821705080433836
take=anyString0.821705080433836

```

图 10-35 成功传输数据

10.3.6 类 DelayQueue 的使用

类 DelayQueue 提供一种延时执行任务的队列。

创建名称为 DelayQueueTest1 的项目，类 Userinfo.java 代码如下：

```
package entity;

import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;

public class Userinfo implements Delayed {

    private long delayNanoTime; // 延迟的纳秒
    private String username;

    public Userinfo(long delayTime, String username) {
        super();
        this.username = username;

        TimeUnit unit = TimeUnit.SECONDS;
        delayNanoTime = System.nanoTime() + unit.toNanos(delayTime);
    }

    public String getUsername() {
        return username;
    }

    @Override
    public int compareTo(Delayed o) {
        if ((this.getDelay(TimeUnit.NANOSECONDS) - o
            .getDelay(TimeUnit.NANOSECONDS)) < 0) {
            return -1;
        }
        if ((this.getDelay(TimeUnit.NANOSECONDS) - o
            .getDelay(TimeUnit.NANOSECONDS)) > 0) {
            return 1;
        }
        return 0;
    }

    public long getDelayNanoTime() {
        return delayNanoTime;
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return unit.convert(delayNanoTime - System.nanoTime(),
            TimeUnit.NANOSECONDS);
    }
}
```

类 Test1.java 代码如下：

```

package test;

import java.util.concurrent.DelayQueue;

import entity.UserInfo;

public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        DelayQueue<UserInfo> queue = new DelayQueue<UserInfo>();
        queue.add(new UserInfo(7, "username5"));
        queue.add(new UserInfo(6, "username4"));
        queue.add(new UserInfo(5, "username3"));
        queue.add(new UserInfo(4, "username2"));
        queue.add(new UserInfo(3, "username1"));

        System.out.println(" " + System.currentTimeMillis());

        System.out.println(queue.take().getUsername() + " "
            + System.currentTimeMillis());
        System.out.println(queue.take().getUsername() + " "
            + System.currentTimeMillis());
        System.out.println(queue.take().getUsername() + " "
            + System.currentTimeMillis());
        System.out.println(queue.take().getUsername() + " "
            + System.currentTimeMillis());
        System.out.println(queue.take().getUsername() + " "
            + System.currentTimeMillis());
    }
}

```

程序运行结果如图 10-36 所示。

任务被成功延迟运行。

	1438329713796
username1	1438329716796
username2	1438329717796
username3	1438329718796
username4	1438329719796
username5	1438329720796

图 10-36 运行结果

10.3.7 类 LinkedTransferQueue 的使用

类 LinkedTransferQueue 提供的功能与 SynchronousQueue 有些类似，但其具有嗅探功能，也就是可以尝试性地添加一些数据。

1. 方法 take 的测试

类 LinkedTransferQueue 中的 take() 方法也具有阻塞特性。创建名称为 LinkedTransferQueue_1 的项目，类 MyServiceA.java 代码如下：

```

package test1;

import java.util.concurrent.LinkedTransferQueue;
import java.util.concurrent.TransferQueue;

```

```
public class MyServiceA {
    public TransferQueue queue = new LinkedTransferQueue();
}
```

类 ThreadA.java 代码如下：

```
package test1;

public class ThreadA extends Thread {
    private MyServiceA service;

    public ThreadA(MyServiceA service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " begin "
                + System.currentTimeMillis());
            System.out.println("取得的值：" + service.queue.take());
            System.out.println(Thread.currentThread().getName() + " end "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 Test1.java 代码如下：

```
package test1;

public class Test1 {
    public static void main(String[] args) {
        MyServiceA service = new MyServiceA();
        ThreadA a = new ThreadA(service);
        a.start();
    }
}
```

程序运行结果呈阻塞状态，因为没有数据可供获取，效果如图 10-37 所示。

2. 方法 transfer(e) 的使用：测试 1

方法 transfer(e) 的作用为：

1) 如果当前存在一个正等待获取值的消费者线程，则把数据立即传输过去；

2) 否则会将元素插入到队列的尾部，并且进入阻塞状态，直到有消费者线程取走该元素。

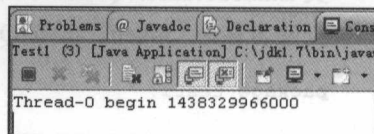


图 10-37 阻塞了

创建项目 `LinkedTransferQueue_2`，先来测试第 2 种情况。

类 `MyServiceB.java` 代码如下：

```
package test2;

import java.util.concurrent.LinkedTransferQueue;
import java.util.concurrent.TransferQueue;

public class MyServiceB {
    public TransferQueue queue = new LinkedTransferQueue();
}
```

类 `ThreadB2.java` 代码如下：

```
package test2;

public class ThreadB2 extends Thread {
    private MyServiceB service;

    public ThreadB2(MyServiceB service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " beginB "
                + System.currentTimeMillis());
            service.queue.transfer("我从 ThreadB2 来");
            System.out.println(Thread.currentThread().getName() + " endB "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 `Test2.java` 代码如下：

```
package test2;

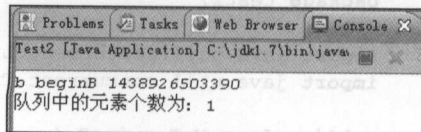
public class Test2 {
    public static void main(String[] args) {
        try {
            MyServiceB service = new MyServiceB();

            ThreadB2 b = new ThreadB2(service);
            b.setName("b");
            b.start();
        }
    }
}
```

```

        Thread.sleep(3000);
        System.out.println(" 队列中的元素个数为: " + service.queue.size());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```



程序运行结果如图 10-38 所示。

图 10-38 队列中有 1 个数据但没有消费者来取

3. 方法 transfer(e) 的使用：测试 2

继续测试第 1 种情况。如果当前存在一个正等待获取值的消费者线程，则把数据立即传输过去。

创建测试用的项目 LinkedTransferQueue_3，类 MyServiceB.java 代码如下：

```

package test2;

import java.util.concurrent.LinkedTransferQueue;
import java.util.concurrent.TransferQueue;

public class MyServiceB {
    public TransferQueue queue = new LinkedTransferQueue();
}

```

类 ThreadB1.java 代码如下：

```

package test2;

public class ThreadB1 extends Thread {
    private MyServiceB service;

    public ThreadB1(MyServiceB service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " beginA "
                + System.currentTimeMillis());
            System.out.println(" 取得的值: " + service.queue.take());
            System.out.println(Thread.currentThread().getName() + " endA "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

类 ThreadB2.java 代码如下:

```
package test2;

public class ThreadB2 extends Thread {

    private MyServiceB service;

    public ThreadB2(MyServiceB service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " beginB "
                + System.currentTimeMillis());
            service.queue.transfer("我从 ThreadB2 来");
            System.out.println(Thread.currentThread().getName() + " endB "
                + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 Test2.java 代码如下:

```
package test2;

public class Test2 {

    public static void main(String[] args) {
        try {
            MyServiceB service = new MyServiceB();

            ThreadB1 a = new ThreadB1(service);
            a.setName("a");
            ThreadB2 b = new ThreadB2(service);
            b.setName("b");

            a.start();
            Thread.sleep(4000);
            b.start();

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 10-39 所示。

4. 方法 tryTransfer(e) 的使用

方法 tryTransfer(e) 的作用为：

1) 如果当前存在一个正在等待获取的消费者线程，使用 tryTransfer(e) 方法会立即传输数据；

2) 否则，如果不存在，则返回 false，并且数据不放入队列中，执行的效果是不阻塞的。

第 1 点已经在前面的章节测试过了，测试一下第 2 点。

创建测试用的项目 LinkedTransferQueue_4，类 MyServiceB.java 代码如下：

```
package test2;

import java.util.concurrent.LinkedTransferQueue;
import java.util.concurrent.TransferQueue;

public class MyServiceB {
    public TransferQueue queue = new LinkedTransferQueue();
}
```

类 ThreadB1.java 代码如下：

```
package test2;

public class ThreadB1 extends Thread {

    private MyServiceB service;

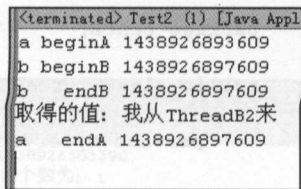
    public ThreadB1(MyServiceB service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " beginA "
            + System.currentTimeMillis());
        System.out.println("tryTransfer(e) 返回值为: "
            + service.queue.tryTransfer("我是数据"));
        System.out.println(Thread.currentThread().getName() + "    endA "
            + System.currentTimeMillis());
    }
}
```

类 Test2.java 代码如下：

```
package test2;

public class Test2 {
```



```
<terminated> Test2 (1) [Java Appl
a beginA 1438926893609
b beginB 1438926897609
b    endB 1438926897609
取得的值: 我从ThreadB2来
a    endA 1438926897609
```

图 10-39 消费者先出现后传输数据


```

public static void main(String[] args) {
    try {
        MyServiceB service = new MyServiceB();
        ThreadB1 a = new ThreadB1(service);
        a.setName("a");

        a.start();
        Thread.sleep(4000);
        System.out.println(" 队列大小为: " + service.queue.size());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

```

a beginA 1438927381406
tryTransfer(e) 返回值为: false
a   endA 1438927381406
队列大小为: 0

```

程序运行结果如图 10-40 所示。

图 10-40 运行结果

5. 方法 tryTransfer(E e, long timeout, TimeUnit unit) 的使用

方法 tryTransfer(E e, long timeout, TimeUnit unit) 的作用为:

- 1) 如果当前存在 1 个正在等待获取数据的消费者线程, 则立即将数据传输给它;
- 2) 否则将把元素插入到队列尾部, 等待被消费者线程获取消费掉;
- 3) 如果在指定的时间内元素没有被消费者线程获取, 则返回 false, 并且将元素从队列中移除。

前面 2 点在前面章节已经验证过了, 下面验证一下第 3 点。

创建测试用的项目 LinkedTransferQueue_5, 类 MyServiceB.java 代码如下:

```

package test2;

import java.util.concurrent.LinkedTransferQueue;
import java.util.concurrent.TransferQueue;

public class MyServiceB {
    public TransferQueue queue = new LinkedTransferQueue();
}

```

类 ThreadB1.java 代码如下:

```

package test2;

import java.util.concurrent.TimeUnit;

public class ThreadB1 extends Thread {
    private MyServiceB service;

    public ThreadB1(MyServiceB service) {

```

```

    super();
    this.service = service;
}

@Override
public void run() {
    try {
        System.out.println(Thread.currentThread().getName() + " beginA "
            + System.currentTimeMillis());
        System.out.println(" 返回值为: "
            + service.queue.tryTransfer("我是元素", 5, TimeUnit.SECONDS));
        System.out.println(Thread.currentThread().getName() + "  endA "
            + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

类 Test2.java 代码如下:

```

package test2;

public class Test2 {

    public static void main(String[] args) {
        try {
            MyServiceB service = new MyServiceB();

            ThreadB1 a = new ThreadB1(service);
            a.setName("a");
            a.start();

            Thread.sleep(500);

            System.out.println("A处 队列大小: " + service.queue.size());

            Thread.sleep(8000);

            System.out.println("A处 队列大小: " + service.queue.size());

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

```

<terminated> Test2 (3) [Java Appl
a beginA 1438927756593
A处 队列大小: 1
返回值为: false
a  endA 1438927761609
A处 队列大小: 0

```

程序运行结果如图 10-41 所示。

图 10-41 超时自动删除了

6. 方法 boolean hasWaitingConsumer() 和 int getWaitingConsumerCount() 的测试

方法 boolean hasWaitingConsumer() 的作用是判断有没有消费者在等待数据, 方法 int

getWaitingConsumerCount() 的作用是取得有多少个消费者在等待数据。

创建测试用的项目 LinkedTransferQueue_6。

类 MyServiceC.java 代码如下：

```
package test3;

import java.util.concurrent.LinkedTransferQueue;
import java.util.concurrent.TransferQueue;

public class MyServiceC {
    public TransferQueue queue = new LinkedTransferQueue();
}
```

类 ThreadC.java 代码如下：

```
package test3;

public class ThreadC extends Thread {

    private MyServiceC service;

    public ThreadC(MyServiceC service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " 取得的值: "
                + service.queue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 Test3.java 代码如下：

```
package test3;

public class Test3 {

    public static void main(String[] args) throws InterruptedException {
        MyServiceC service = new MyServiceC();

        for (int i = 0; i < 10; i++) {
            ThreadC a = new ThreadC(service);
            a.setName("a");
            a.start();
        }
    }
}
```

```

Thread.sleep(1000);
System.out
    .println(" 有没有线程正在等待数据？ " + service.queue.hasWaitingConsumer());
System.out.println(" 有 " + service.queue.getWaitingConsumerCount()
    + " 个线程正在等待数据 ");
}
}

```

程序运行结果如图 10-42 所示。

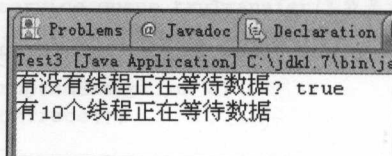
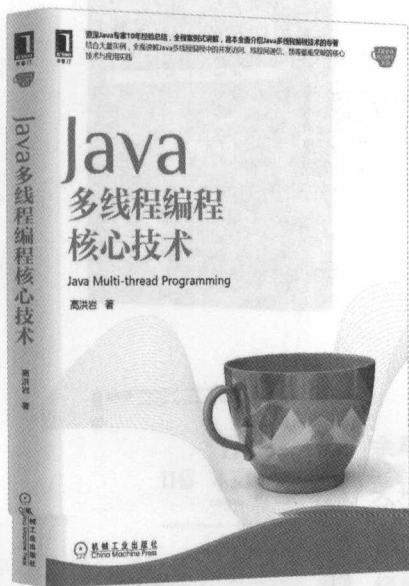


图 10-42 获取有多少个消费者在等待数据

10.4 本章总结

本章主要介绍了 Java 并发包中的集合框架，集合在使用 Java 语言中是非常重要的技能点，而并发集合框架在原来功能的基础上进行再次强化，完全支持多线程环境下的数据处理，大大提高了开发效率，有效保证了数据的存储结构，以常见的阻塞与非阻塞算法加强并发包中功能的可用性。

推荐阅读



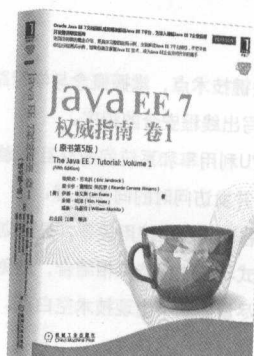
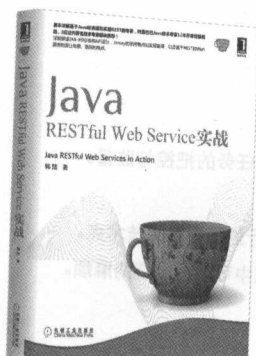
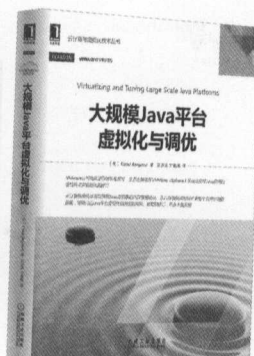
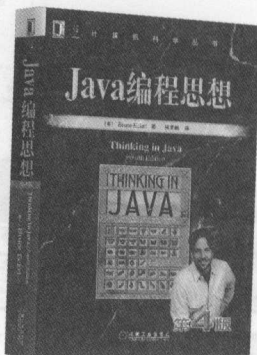
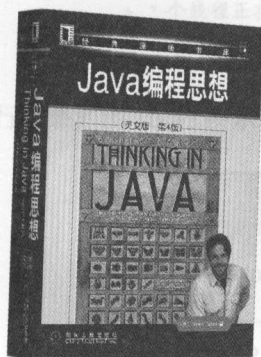
Java多线程编程核心技术

作者：高洪岩 书号：978-7-111-50206-7 定价：69.00元

内容亮点：

- 线程类的核心API的使用与关键技术点，掌握概念与学习路径。
- 并发访问控制技术，即如何写出线程安全的程序。
- 线程间通信技术，以提高CPU利用率和系统间的交互，增强对线程任务的把控与监督。
- Lock对象技术，以更好实现并发访问时的同步处理。
- 定时器类中的多线程技术，移动开发中使用较多，是计划/任务执行里很重要的技术点。
- 如何安全、正确地将单例模式与多线程技术相结合，避免实际应用中可能会出现麻烦。
- 多个查疑补漏的技术案例，尽量做到不出现技术空白点。

推荐阅读



作者简介



高洪岩

某世界500强企业高级项目经理，10余年项目管理与开发经验，10年Java相关开发经验，深谙Java技术开发难点与要点，拥有良好的技术素养和丰富的实践经验。精通J2EE核心技术、基于EJB的分布式系统开发、Android移动开发、智能报表、多线程及高并发等相关的技术内容，近期继续关注并发相关的前沿技术。著有技术畅销书《Java多线程编程核心技术》，喜欢将技术与教育相结合的方式共享知识，得以共同提高。生活中喜欢摄影，对轮滑，旅游，航模亦兴趣浓厚。

Java

并发编程

核心方法与框架

Java并发编程无处不在，服务器、数据库、应用，Java并发是永远不可跳过的沟坎，优秀的程序员一定要在Java并发领域进行炼狱式的学习，吸收消化并最终转化成软件产品成果。另外，单纯从Java程序员成长计划这方面进行考虑，Java多线程/并发也依然是想深入学习Java必须要掌握的技术，比如在软件公司中接触的“缓存”，“分布式一致性”，“高并发框架”，“海量数据处理”，“高效订单处理”等都与Java多线程、Java并发紧密相关。进行大数据、分布式、高并发类的专题攻克时，并发编程的学习必不可少，但并发编程学习曲线陡峭，多弯路和“坑”。本书基本完全覆盖了Java并发包中核心类、API与并发框架，最大程度介绍了每个常用类的使用，以案例的方式进行讲解，以使读者快速学习，迅速掌握。

本书有以下特点：

- 不留遗漏——全面覆盖Java并发知识点；
- 直击要害——实战化案例，精准定位技术细节；
- 学以致用——精要式演示，确保开发/学习不脱节；
- 潜移默化——研磨式知识讲解，参透技术要点；
- 提升效率——垂直式技术精解，不绕弯路；
- 循序提升——渐进式知识点统排，确保连贯。



投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机\程序设计\Java

ISBN 978-7-111-53521-8



9 787111 535218 >

定价：79.00元